

WebSphere eXtreme Scale Best Practices for Operation and Management

Tips for capacity planning

Leading practices for operations

Grid configuration



Ying Ding
Bertrand Fayn
Art Jolin
Hendrik Van Run
Carla Sadtler
Chunmo Son
Sukumar Subburaj
Tong Xie

Redbooks



International Technical Support Organization

**WebSphere eXtreme Scale Best Practices for
Operation and Management**

August 2011

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

First Edition (August 2011)

This edition applies to WebSphere eXtreme Scale V7.1.

© Copyright International Business Machines Corporation 2011. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contact an IBM Software Services Sales Specialist



Start SMALL, Start BIG, ... **JUST START**

architectural knowledge, skills, research and development . . .
that's IBM Software Services for WebSphere.

Our highly skilled consultants make it easy for you to design, build, test and deploy solutions, helping you build a smarter and more efficient business. **Our worldwide network of services specialists wants you to have it all!** Implementation, migration, architecture and design services: IBM Software Services has the right fit for you. We also deliver just-in-time, customized workshops and education tailored for your business needs. You have the knowledge, now reach out to the experts who can help you extend and realize the value.

For a WebSphere services solution that fits your needs, contact an IBM Software Services Sales Specialist:
ibm.com/developerworks/websphere/services/contacts.html

Contents

Contact an IBM Software Services Sales Specialist	iii
Notices	ix
Trademarks	x
Preface	xi
The team who wrote this book	xi
Now you can become a published author, too!	xiii
Comments welcome.	xiii
Stay connected to IBM Redbooks	xiii
Chapter 1. Introduction to WebSphere eXtreme Scale	1
1.1 Benefits of using WebSphere eXtreme Scale	2
1.2 Adoption of WebSphere eXtreme Scale	2
1.3 Common usage patterns for WebSphere eXtreme Scale	4
1.3.1 Side cache	6
1.3.2 Inline cache	7
1.3.3 Application state store	8
1.3.4 Extreme transaction processing (XTP)	9
1.4 The contents of this book	11
Chapter 2. WebSphere eXtreme Scale architecture	13
2.1 A grid from a user's point of view	14
2.2 Grid component overview	15
2.3 A grid from WebSphere eXtreme Scale's view	17
2.3.1 Catalog service	17
2.3.2 Shards	18
2.4 Client grid access	22
2.5 WebSphere eXtreme Scale internal components	22
2.5.1 Session	22
2.5.2 Map	23
2.5.3 ObjectMap	23
2.5.4 Tuples	23
2.5.5 Backing maps	24
2.5.6 Grid clients and backing maps	25
Chapter 3. Topology and infrastructure	27
3.1 Topology design options	28
3.1.1 Understanding WebSphere eXtreme Scale topology	28
3.1.2 Stand-alone vs. managed installations	29
3.1.3 Local, embedded, and distributed grid topologies	30
3.1.4 Multiple data center topologies: Zone vs. multi-master replication	31
3.2 WebSphere eXtreme Scale distributed topology	32
3.2.1 Catalog service placement	33
3.2.2 Example	35
3.3 Zone-based topology	36
3.3.1 When to use zone-based topology	38
3.3.2 Synchronous and asynchronous replication	38
3.3.3 The placement of catalog servers	38

3.3.4 Proximity-based routing	38
3.4 Multi-master replication topology	39
3.4.1 Topology options.	39
3.4.2 Performance and fault tolerance considerations.	40
3.4.3 Collision management considerations	41
3.4.4 Testing the topology	42
3.5 Port assignments and firewall rules	42
3.5.1 Catalog service domain ports	43
3.5.2 Container server ports	45
3.5.3 Client configuration	46
3.5.4 Multi-master replication ports	46
3.5.5 SSL ports	47
3.5.6 Summary.	47
Chapter 4. Capacity planning and tuning	49
4.1 Planning for capacity	50
4.1.1 Planning for catalog servers	50
4.1.2 Planning for container servers: Building blocks	50
4.1.3 What do we need to size when performing capacity planning	51
4.1.4 Calculating the required memory for the data in the grid	52
4.1.5 Determining the maximum heap size and physical memory per JVM	54
4.1.6 Topology considerations for failure handling.	58
4.1.7 Determining the number of partitions	58
4.1.8 Determining numInitialContainers.	60
4.1.9 Determining the number of CPUs.	60
4.1.10 An example of a WebSphere eXtreme Scale sizing exercise.	61
4.2 Tuning the JVM.	63
4.2.1 Selecting a JVM for performance	63
4.2.2 Tuning for efficient garbage collection	63
4.2.3 Increasing the ORB thread pool	66
4.2.4 Thread count.	66
4.2.5 Sources and references	67
Chapter 5. Grid configuration	69
5.1 Configuration overview	70
5.1.1 Server XML configuration	71
5.1.2 Client XML configuration.	72
5.1.3 Server properties	74
5.1.4 Client properties	76
5.1.5 Externalizing the server XML configuration in WebSphere Application Server.	77
5.1.6 Duplicate server names in WebSphere Application Server	83
5.2 Catalog service domain.	83
5.2.1 Configuring a catalog service domain in a WebSphere environment	84
5.3 ObjectGrid plug-ins	87
5.3.1 The TransactionCallback plug-in.	87
5.3.2 The ObjectGridEventListener plug-in	91
5.4 BackingMap plug-ins.	94
5.4.1 Loaders: Choices and configurations	95
5.4.2 OptimisticCallback plug-ins.	104
5.4.3 MapEventListener plug-ins	106
5.4.4 Indexing plug-ins.	107
Chapter 6. Performance planning for application developers	109
6.1 Copy mode method preferred practices	110

6.1.1	COPY_ON_READ_AND_COMMIT mode	112
6.1.2	COPY_ON_READ mode.	112
6.1.3	COPY_ON_WRITE mode.	113
6.1.4	NO_COPY mode	113
6.1.5	COPY_TO_BYTES mode.	113
6.2	Evictor performance preferred practices	115
6.2.1	Default (time-to-live) evictor	116
6.2.2	Pluggable evictors with LFU and LRU properties	117
6.2.3	Memory-based eviction.	120
6.3	Locking performance preferred practices	122
6.3.1	Pessimistic locking strategy	122
6.3.2	Optimistic locking strategy	122
6.3.3	None locking strategy	123
6.3.4	Lock semantics for lockStrategy=PESSIMISTIC, OPTMISTIC, and NONE	123
6.4	Serialization performance	127
6.4.1	A discussion of efficient serialization.	127
6.4.2	Implementing the Externalizable interface	128
6.4.3	Using a custom ObjectTransformer implementation.	130
6.5	Query performance tuning	134
6.5.1	Using parameters	134
6.5.2	Using indexes	134
6.5.3	Using pagination	135
6.5.4	Returning primitive values instead of entities	135
6.5.5	Query plan	135
6.5.6	getPlan method.	135
6.5.7	Query plan trace	136
6.5.8	Query plan examples	136
Chapter 7.	Operations and monitoring	137
7.1	Starting and stopping WebSphere eXtreme Scale	138
7.1.1	Starting and stopping catalog services	139
7.1.2	Starting and stopping container servers	140
7.1.3	Performing a full (cold) start of WebSphere eXtreme Scale	142
7.1.4	Performing a partial start of container servers	143
7.1.5	Performing a partial stop of catalog servers	144
7.1.6	Performing a partial stop of container servers	144
7.1.7	Performing a full stop of WebSphere eXtreme Scale	147
7.1.8	What to do when a JVM is lost	148
7.2	The placement service in WebSphere eXtreme Scale	149
7.2.1	Where the placement service runs	149
7.2.2	Quorum and the placement service	150
7.2.3	When and how the placement service runs	153
7.3	The xsadmin command-line tool	155
7.3.1	Using xsadmin in an embedded environment.	156
7.3.2	Useful xsadmin commands.	157
7.4	Configuring failure detection	162
7.4.1	Container failover detection	163
7.4.2	Client failure detection	166
7.5	Monitoring WebSphere eXtreme Scale.	167
7.5.1	Operating system monitoring	168
7.5.2	Monitoring WebSphere eXtreme Scale logs.	169
7.5.3	WebSphere eXtreme Scale web console	171
7.5.4	Monitoring with Tivoli Performance Viewer.	178

7.5.5 Monitoring using a WebSphere eXtreme Scale ping client	180
7.5.6 Additional monitoring tools	181
7.6 Applying product updates	181
7.6.1 Overview	182
7.6.2 Procedures	182
Appendix A. Sample code	187
FastSerializedKeyOrEntry_Externalizable.java	188
Appendix B. Additional material	197
Locating the Web material	197
Using the Web material	197
Downloading and extracting the Web material	198
Related publications	199
IBM Redbooks	199
Online resources	199
Help from IBM	200

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:


This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®
DataPower®
IBM®
MVS™

Rational®
Redbooks®
Redbooks (logo) ®
Tivoli®

WebSphere®
z/OS®

The following terms are trademarks of other companies:

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Preface

This IBM® Redbooks® publication contains a summary of the leading practices for implementing and managing a WebSphere® eXtreme Scale installation. The information in this book is a result of years of experience that IBM has had in with production WebSphere eXtreme Scale implementations. The input was received from specialists, architects, and other practitioners who have participated in engagements around the world.

The book provides a brief introduction to WebSphere eXtreme Scale and an overview of the architecture. It then provides advice about topology design, capacity planning and tuning, grid configuration, ObjectGrid and backing map plug-ins, application performance tips, and operations and monitoring.

This book is written for a WebSphere eXtreme Scale-knowledgeable audience.

The team who wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.

Ying Ding is an IT veteran with 18 years of industry experience. He has worked for Bank of America as a senior technology manager. His team there supported large WebSphere systems on many platforms from mainframe to mid-range servers. Now, Ding works for a large international retailer as a middleware engineering manager focusing on WebSphere technologies. Ding has worked for BSG Consulting, taught computer science at Houston Community College, and was with IBM IGS as a consultant to major IBM energy and utilities clients throughout the Houston area. An IBM Certified WebSphere System Expert and IBM Certified E-Business Technologist, he holds a masters degree in Computer Science. He leads the Charlotte, N.C. WebSphere User Group. Ying Ding is a prolific author on WebSphere engineering, large system stability, and the IT engineering process.

Bertrand Fayn is an IT Consultant working in France for ISSW. He has five years of experience in WebSphere Application Server Network Deployment (ND). His areas of expertise include development, administration, problem determination, and performance tuning of WebSphere Application Server. He started to work on various WebSphere eXtreme Scale projects three years go. He holds a degree in Physics and Chemistry from the University of Lannion.

Art Jolin is a software architect, developer, consultant, and instructor with over 34 years experience in the field. Art has extensive experience in OO architecture, design, and development in Java, C++, and Smalltalk and has been a team lead for major projects both within IBM and as a consultant. Art also has extensive experience with the WebSphere extended family of products, especially WebSphere Application Server, WebSphere eXtreme Scale, and Rational Application Developer. Art currently works as an IBM Certified IT Specialist consultant for ISS Federal, IBM, and has consulted on WebSphere eXtreme Scale almost exclusively since January 2008.

Hendrik Van Run is an IBM Certified Consulting IT Specialist within the Software Services for WebSphere team in Hursley, UK. He holds a Masters degree in Physics from Utrecht University in the Netherlands. Hendrik advised many of the IBM enterprise clients on issues, such as high availability, performance, and queuing problems across the WebSphere

Application Server product families. He also specialized in the WebSphere eXtended Deployment suite of products. More recently, he started working with a number of IBM clients on WebSphere eXtreme Scale implementations. Prior to joining the Software Services for WebSphere team, Hendrik worked for IBM Global Services in the Netherlands. He has worked with WebSphere Application Server since Version 3.5 and is a co-author of two other IBM Redbooks publications.

Carla Sadtler is a Consulting IT Specialist at the ITSO, Raleigh Center. She writes extensively about WebSphere products and solutions. Before joining the ITSO in 1985, Carla worked in the Raleigh branch office as a Program Support Representative, supporting MVS™ clients. She holds a degree in Mathematics from the University of North Carolina at Greensboro.

Chunmo Son is a WebSphere IT Specialist with over 18 years of deep technical experience, the last 12 with IBM. An early evangelist of WebSphere to ISVs in the Silicon Valley area, he also worked with the WebSphere High Performance team in the Silicon Valley Lab for high profile clients on performance-related projects. Later, he moved to the NA proof of concept (POC) team where he has been instrumental in numerous WebSphere solution wins through his work conducting business process management (BPM) and Infrastructure POCs at clients throughout the geography. Chunmo recently moved full-time into his worldwide role where he focuses on elastic caching with WebSphere eXtreme Scale and the WebSphere DataPower® XC10 Appliance capabilities, as well as other core parts of the WebSphere Foundation portfolio.

Sukumar Subburaj (Suku) is a Software Consultant working for IBM Software Labs, India. His areas of expertise include WebSphere Application Infrastructure design, development, and management on System z and distributed platforms. Currently, he works on WebSphere Virtual Enterprise and WebSphere eXtreme Scale projects. He is also part of the infrastructure practices team in India Software Lab Services and involved in WebSphere infrastructure and performance projects. He holds a Bachelor of Engineering degree in Electronics and Communication Engineering (ECE) from Anna University, India.

Tong Xie is an IBM Certified IT Specialist and works as a Senior Architect for a Global Integrated Account in Software Group, IBM China. He has 18 years of IT experience, with six of those years working as above-country roles on complex WebSphere Portal deployment, managing Early Adoption Programs and designing solutions based on WebSphere Software for Telecom (WST) for clients across Asia Pacific. He spent most of his career in the Telecommunication and Finance industries. He has working experience in mainland China, Hong Kong, Australia, and the United States. He has worked at IBM for 10 years. He holds a bachelor degree in Computer Science from Zhejiang University, China.

Thanks to the following people for their contributions to this project:

Margaret Ticknor
International Technical Support Organization, Raleigh Center

Ann Black
Billy Newport
Chris Johnson
Michael Smith
IBM US

Jian Tang
IBM China

Ek Chotechawanwong
IBM Thailand

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and client satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks form found at:

ibm.com/redbooks

- ▶ Send your comments in an email to:

redbooks@us.ibm.com

- ▶ Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- ▶ Find us on Facebook:

<http://www.facebook.com/IBMRedbooks>

- ▶ Follow us on Twitter:

<http://twitter.com/ibmredbooks>

- ▶ Look for us on LinkedIn:

<http://www.linkedin.com/groups?home=&gid=2130806>

- ▶ Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>

- ▶ Stay current on recent Redbooks publications with RSS Feeds:

<http://www.redbooks.ibm.com/rss.html>



Introduction to WebSphere eXtreme Scale

This chapter outlines the scalability and performance challenges that IBM WebSphere eXtreme Scale can address in today's dynamic business and IT environments. It also provides an introduction to common usage patterns of WebSphere eXtreme Scale.

This chapter includes the following topics:

- ▶ Benefits of using WebSphere eXtreme Scale
- ▶ Adoption of WebSphere eXtreme Scale
- ▶ Common usage patterns for WebSphere eXtreme Scale
- ▶ The contents of this book

1.1 Benefits of using WebSphere eXtreme Scale

Today's dynamic business environment and economic uncertainty mean organizations must work smarter to remain competitive and to respond to changing customer demands. The key to working smarter is business agility and cost optimization. Organizations need to ensure that crucial applications can meet the requirements of rapid increases in demand, can deliver immediate and consistent responses, and can scale as necessary. There is also a need for enhanced application performance to keep pace with rising customer expectations regarding application response time while also managing the costs of the infrastructure.

Many challenges in meeting these demands center around data access. A standard three-tier application will typically present the following challenges:

- ▶ Keeping system response time low while user load increases
- ▶ Dealing with a high database load that causes slow data access
- ▶ Scaling the current infrastructure as the data volume grows
- ▶ Dealing with a volume of data that is so large that it cannot be stored in a single, physical system
- ▶ Maintaining the state in memory and replicating that state across multiple systems in various data centers in case of failover

The solution to these challenges can be found with WebSphere eXtreme Scale. WebSphere eXtreme Scale is an elastic, scalable, in-memory data grid. It allows business applications to process billions of transactions per day with efficiency and near-linear scalability. The data grid dynamically caches, partitions, replicates, and manages application data and business logic across multiple servers and virtualization environments. With WebSphere eXtreme Scale, you can also get qualities of service, such as transactional integrity, high availability, and predictable response time.

1.2 Adoption of WebSphere eXtreme Scale

WebSphere eXtreme Scale provides a data grid to store data. How you access that grid, and the type of data that you store in it can vary depending on the application. A detailed systems and design analysis is required to determine if, when, and how you use WebSphere eXtreme Scale. This analysis is an important exercise for devising a decision tree road map for adoption of any new technology.

Figure 1-1 illustrates a decision tree that can help in this evaluation.

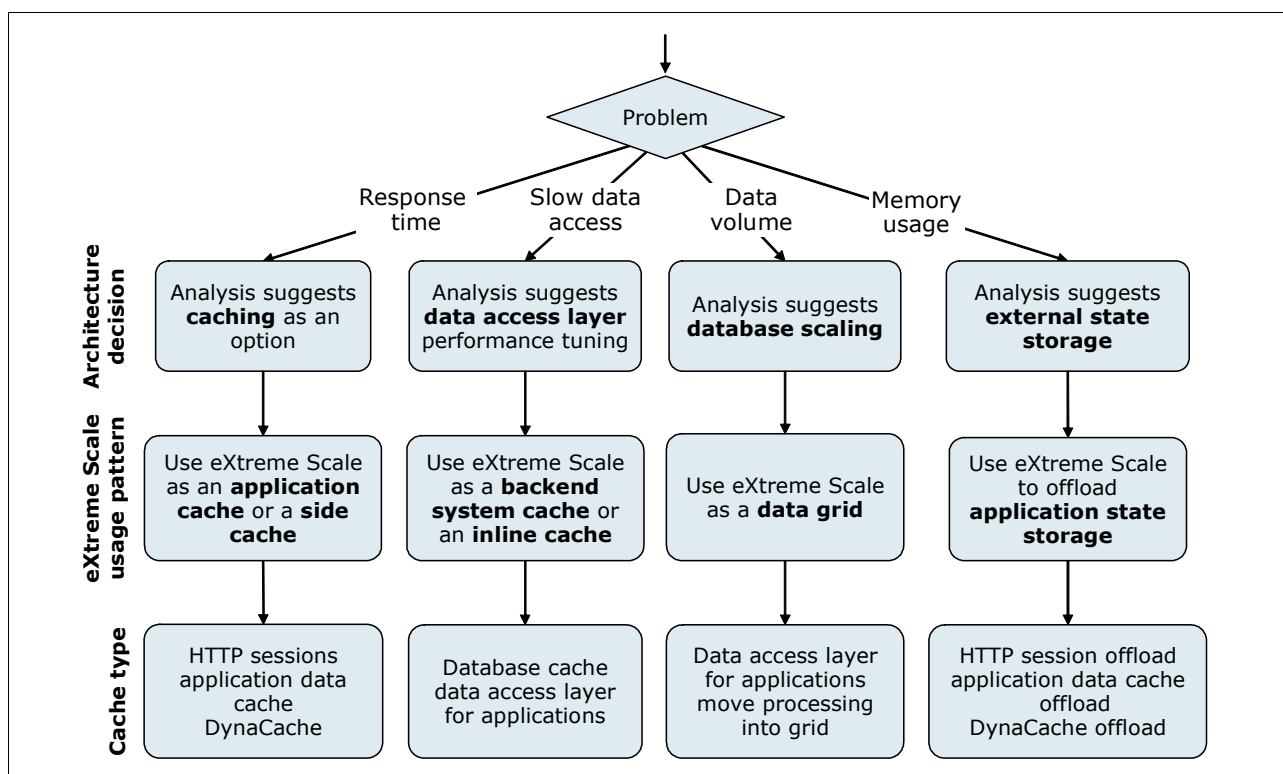


Figure 1-1 Decision tree for adopting WebSphere eXtreme Scale

The decision tree addresses the issues of inadequate response time, slow data access, data volume issues that affect performance, and memory usage problems. Based on the analysis of similar problems, the first tier in the tree suggests the type of solution that is appropriate at an architectural level. The second tier in the tree suggests the usage pattern on how WebSphere eXtreme Scale can be used to implement the solution. The third tier provides information about the specific types of caching that might prove useful in resolving the problem.

The problem categories shown in the decision tree are interrelated, in that high database load leads to slow data access, which by itself is a cause of slow application response times. Removing one scalability bottleneck by introducing caching or a more advanced data grid solution often reveals another bottleneck. It might take a few iterations and a few uses of WebSphere eXtreme Scale before the application meets its scalability targets.

The problems shown in the decision tree invite a number of incremental solutions, such as tuning database settings for greater performance, optimizing queries, and scaling up database servers. These solutions do not address the greater scalability limitations of the application but rather move it toward the limits of the current architecture. The effectiveness of these solutions depends on how inefficient the application currently is and the availability of more powerful server equipment, both of which can quickly reach the point of diminishing returns.

1.3 Common usage patterns for WebSphere eXtreme Scale

The following usage patterns are common for WebSphere eXtreme Scale:

- ▶ Side cache
- ▶ Inline cache
- ▶ Application state store
- ▶ eXtreme Transaction Processing (XTP) or simply a data grid

Table 1-1 summarizes these usage patterns.

Table 1-1 Summary of WebSphere eXtreme Scale usage patterns

Context	Problem	Pattern or Solution	Results
The application is coded to read data from the database	The disk I/O is an expensive operation. When the workload is up, we encounter slow response time due to the large number of disk I/Os to retrieve data from the back-end database.	Side cache: A <i>cache</i> is normally a component in memory to store data. When an application needs data, it will look for data in the cache first. If the data is in the cache (cache hit), the data is simply returned to the application. If the data is not in the cache (cache miss), the data is retrieved from the back-end database and added into cache. The application talks to both the side cache and the database.	<ul style="list-style-type: none">▶ Reduced response time: The more requests can be served from the cache, the faster the overall system performance is.▶ Reduced load on database. Off-load redundant processing.▶ Can be used as:<ul style="list-style-type: none">– Hibernate/OpenJPA L2 cache– Dynamic Cache service provider– Service-oriented architecture (SOA) state store– SOA result cache

Context	Problem	Pattern or Solution	Results
The application is coded to read data from and write data to the database	Write operations are normally slower than read operations when accessing a database. When there are many create, update, or delete transactions in the application, writing to the database becomes the bottleneck of the whole system.	Inline cache: When an application needs data, it interacts only with the inline cache. The cache has a loader to move the data between the cache and the back-end database. There are three scenarios with inline cache: read-through, write-through, and write behind: <ul style="list-style-type: none"> ▶ A read-through cache is a read-only sparse cache that lazily loads data entries by key as they are requested, without requiring the caller to know how the entries are populated. ▶ With a write-through cache, you have the read-through behavior, but for writes, the transaction does not complete until the data is written to the backing store. ▶ With a write-behind cache, the transaction completes when the data is written to the cache. The cache uses replication to keep the data available until it is written to the backing store. 	<ul style="list-style-type: none"> ▶ Loaders used to integrate with an existing back-end database ▶ Read through reads at cache speed if the data is already in cache ▶ Write through write operation to the cache is not complete until the data is synchronized to back-end database ▶ Write-behind pushes changes asynchronously from cache to back-end database ▶ Reduced load on database
Application state needs to be replicated across multiple systems and some times even multiple data centers	Application state can be a lot of things, including business data, events, tables, maps, images, session information, and much more. When the user base increases, so does the memory consumption to store the state. You might reach a point where you cannot allocate more system resources to keep up with the growing number of concurrent users.	Application state store: An application state store provides large in-memory data grids. This pattern is similar to the inline cache usage pattern. Where the inline cache is focused on the database, this pattern is geared toward a large, general data-in-memory store. Application state store can be used for: <ul style="list-style-type: none"> ▶ HTTP session store ▶ Application session store ▶ Multiple data center support for sessions 	<ul style="list-style-type: none"> ▶ Memory provides faster access times and increased throughput compared to disk and database stores ▶ Reduced response time and increased throughput data in memory ▶ Replication across the grid provides high availability ▶ Write-behind allows for more persistent store when required Moving state replication off the database reduces the load on database.

Context	Problem	Pattern or Solution	Results
When there is more data to be stored in the cache, you need to scale up your caching solution.	Conventional in-memory caches are also limited to only storing what can fit in the free memory of a Java virtual machine (JVM). When we need to cache more than this amount of data, thrashing occurs, where the cache continuously evicts data to make room for other data. You then need to read the required record continually, thereby making the cache useless and exposing the database to the full read load.	Data grid or eXtreme Transaction Processing: eXtreme Transaction Processing (XTP) describes applications that are designed for high performance, reliability, and scale at an efficient cost. XTP applications rely on in-memory processing on grids of low-cost processors. WebSphere eXtreme Scale is used as a distributed caching solution in eXtreme Transaction Processing.	<ul style="list-style-type: none"> ► Support transaction-intense services cost-effectively ► Deliver consistent, predictable responses ► Reduce transaction life-cycle costs ► Be able to handle extreme amounts of business events

In the next sections, we examine these usage patterns in detail.

1.3.1 Side cache

WebSphere eXtreme Scale can be used as a side cache for an application's data access layer. In this usage pattern, the WebSphere eXtreme Scale grid is used to temporarily store objects that normally are retrieved from a back-end database. Applications use the WebSphere eXtreme Scale application programming interfaces (APIs) to check the cache for the desired data. If the data is there, it is returned to the caller. If the data is not there, it is retrieved from the back-end database and inserted into the cache so that the next request can use the cached copy.

The side cache pattern is used when you want the application to have more control in its code. If the application has a common data access layer used by all of the code, the side cache logic can be localized to just that layer without requiring changes throughout the application.

Figure 1-2 illustrates how you can use WebSphere eXtreme Scale as a side cache.

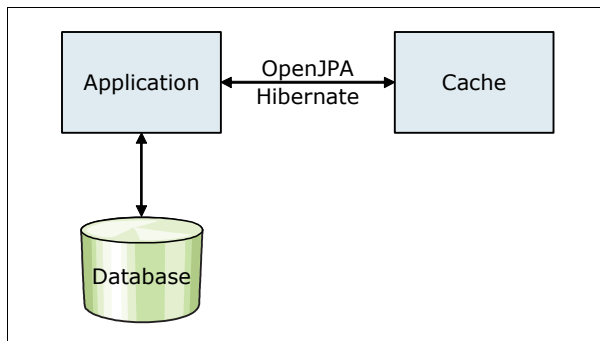


Figure 1-2 Side cache

Using WebSphere eXtreme Scale as a cache provider increases performance when reading and querying data and reduces the load to the database. WebSphere eXtreme Scale has advantages over built-in cache implementations, because the cache is replicated

automatically between all processes. When one client caches a value, all other clients can use the cached value.

1.3.2 Inline cache

When used as an inline cache, WebSphere eXtreme Scale interacts with the back-end database using a loader plug-in. This scenario can simplify data access by allowing applications to access the cache using the WebSphere eXtreme Scale APIs directly. Several caching scenarios are supported to make sure that the data in the cache and the data in the back-end database are synchronized and to avoid “if cache ... else database...” logic in the application code.

Figure 1-2 on page 6 illustrates how an inline cache interacts with the application and back-end database.

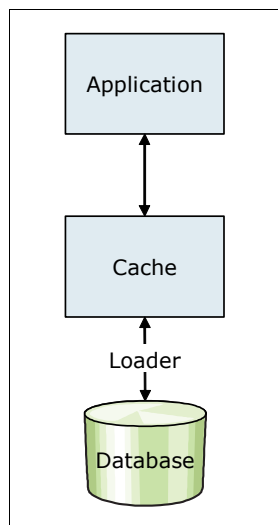


Figure 1-3 Inline cache

The inline caching option simplifies data access, because it allows applications to access the WebSphere eXtreme Scale APIs directly without “if cache ... else database...” logic. WebSphere eXtreme Scale supports the following inline caching scenarios:

- ▶ Read-through
- ▶ Write-through
- ▶ Write-behind

Read-through caching scenario

A *read-through cache* is a read-only sparse cache that lazily loads data entries by key as they are requested, without requiring the caller to know how the entries are populated. If the data cannot be found in the WebSphere eXtreme Scale cache, WebSphere eXtreme Scale retrieves the missing data from the loader plug-in, which loads the data from the back-end database and inserts the data into the cache. Subsequent requests for the same data key are found in the cache until it is removed, invalidated, or evicted.

Write-through caching scenario

A *write-through cache* allows reading (as with read-through caching) and writing. In a write-through cache, every write to the cache synchronously writes to the database using the loader. That is, the write operation will not complete until the cache successfully stores the data to the underlying database. This method provides consistency with the back-end

database but decreases write performance, because the database operation is synchronous. Because the cache and database are both updated, subsequent reads for the same data will be found in the cache, avoiding the database call. A write-through cache is often used in conjunction with a read-through cache.

Write-behind caching scenario

Database synchronization can be improved by writing changes asynchronously. This method is known as a *write-behind* or *write-back* cache. Changes that are normally written synchronously to the loader are instead buffered in WebSphere eXtreme Scale and written to the database using a background thread. Write performance is improved, because the database operation is removed from the client transaction and the database writes can be compressed or in a batch mode.

1.3.3 Application state store

The application state can include business data, events, tables, maps, images, session information, and much more. WebSphere eXtreme Scale can serve as an efficient application state store for applications that make heavy use of session data. For example, you can use WebSphere eXtreme Scale to offload HTTP sessions to the grid, which otherwise are stored and persisted either in memory or in a shared database.

HTTP session store

Many web applications use the HTTP session state management features that are provided by Java Platform, Enterprise Edition (Java EE) application servers. Session state management allows the application to maintain a state for the period of a user's interaction with the application. The traditional example for this session store is a shopping cart where information about intended purchases is stored until the shopping session is completed.

To support high availability and failover, the state information must be available to all application server JVMs. Application servers typically achieve this access by storing the state information in a shared database, or by performing memory-to-memory replication between JVMs. When an HTTP session state is stored in a shared database, scalability is limited by the database server. The disk I/O operations are an expensive performance cost to the application. When the transaction rate exceeds the capacity of the database server, the database server must be scaled up.

When the HTTP session state is replicated in memory between application servers, the limits to scalability vary depending on the replication scheme. Commonly, a simple scheme is used in which each JVM holds a copy of all user session data, so the total amount of state information cannot exceed the available memory of any single JVM. Memory-to-memory replication schemes often trade consistency for performance, meaning that in cases of application server failure or user sessions being routed to multiple application servers, the user experience can be inconsistent and confusing.

To address these challenges, WebSphere eXtreme Scale can be used as a shared in-memory cache for the HTTP session state instead of using a WebSphere traditional shared database or memory-to-memory replication approach. This shared in-memory cache sits in a highly available replicated grid. The grid is not constrained to any one application server product or to any particular management unit, such as WebSphere Application Server Network Deployment cells. User sessions can be shared between any set of application servers, even across data centers, allowing a more reliable and fault-tolerant user session state. No application code change is required when using WebSphere eXtreme Scale to store and manage HTTP session data.

Application session store

There are cases where an application needs to store frequently used Java objects in memory to reduce response time. For example, in a single sign-on (SSO) scenario, you might want to store credential and related information retrieved from an LDAP server in memory. When the number of these frequently used Java objects increases, quite often you cannot store all of them in the memory of a single system. Thus, you need to store the application state in a database. Another example is when you want multiple JVMs, multiple applications, or even separate application server vendors (Tomcat and WebSphere Application Server, for example) to share access to the same Java objects, making it necessary to store them in a location that is accessible to all.

You can use WebSphere eXtreme Scale as a large in-memory data store for application session data. You can linearly scale up as the amount of data grows. Imaging a company with large number of items in their support knowledge base with WebSphere eXtreme Scale, you can implement the “incremental search” or “real-time suggestion” features for customers when searching a specific support document on the website by storing precalculated substring matches in a WebSphere eXtreme Scale grid.

Multiple data center support for sessions

With WebSphere eXtreme Scale zone-based topology and multi-master topology, you can have multiple data center support for sessions. Multiple data centers can be linked together asynchronously, allowing a data center to access data locally and maintain high availability. WebSphere eXtreme Scale allows for rules-based data placement, enabling high availability of the grid due to the redundant placement of data across physical locations. This notion is particularly appealing to enterprise environments that need data replication and availability across geographically dispersed data centers.

1.3.4 Extreme transaction processing (XTP)

The goal of an extreme transaction processing design is to provide secure, large-scale, high-performing transactions across a distributed environment. Extreme transaction processing enables you in building SOA infrastructure for critical applications.

The key concept of extreme transaction processing is to have application code that runs in the grid itself. In the grid, the data is partitioned across the grid servers. When an item is added to the grid, it can be processed by the grid server that stores it, thus moving processing logic to the data.

With large amounts of data, it is much easier to send the logic to the data, so you process the data locally. You do not want to send large amounts of data over the grid.

Inserting a fully populated, partitioned, in-memory cache into the application architecture introduces the possibility of processing the entire data set in parallel at local memory access speeds. Rather than fetching data from the back-end data store and processing it in series, the application can process the data in parallel across multiple grid containers and combine the results together. In general, this process relies on the data items being independent of each other. In cases where the data items are interrelated, the cache must be partitioned so that all related items are placed within the same partition (WebSphere eXtreme Scale allows you to plug in custom logic to control partitioning, via its PartitionableKey interface).

In this scenario, the grid usually becomes the system of record. The database remains as a hardened data store to ensure against data loss caused by catastrophic failure of the systems hosting the grid.

Applying this technique involves changes to the application logic, because the data processing operations must be specified in terms of the parallel processing schemes provided by the grid. As shown in Figure 1-4, the grid becomes a central component of the application architecture, rather than a solution applied at individual points within the application.

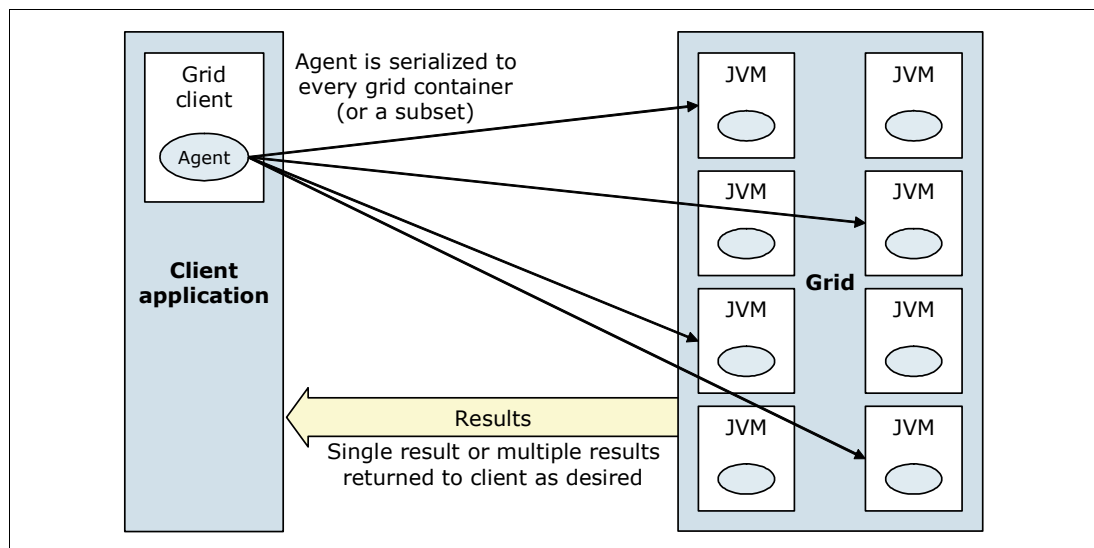


Figure 1-4 Agents sent out to the grid to collocate themselves with the data

An *agent* is code created using the data grid APIs, which provide a vehicle to run logic collocated with the data, to query across partitions, and to aggregate large result sets into a single result set. That is, the agent does the collation of the data into a single unit and returns it to the client.

Two major query patterns exist for data grid applications:

- Parallel map

In this pattern, all or a subset of entries are processed, and a result for each entry processed is returned.

- Parallel reduction

In this pattern, all or a subset of entries are processed, and a single result is calculated for the group of entries.

This approach enables parallel processing of large volumes of data in ways that most likely were not practical without the grid. It might be possible to run processes online that previously were only run as batch jobs. For example, you can calculate per-procedure medical coverage individually for every zip code in the United States by partitioning on zip code and processing multiple zip codes in parallel.

An application using grid agents can be scaled out in terms of both data size and processing capacity by adding more grid containers.

1.4 The contents of this book

The following chapters in the remainder of this book describe the preferred practices in various aspects of WebSphere eXtreme Scale:

- ▶ Chapter 2, “WebSphere eXtreme Scale architecture” on page 13

This chapter provides an architectural view of WebSphere eXtreme Scale. It provides a basis for understanding how the grid stores and manages data and how clients access the data.

- ▶ Chapter 3, “Topology and infrastructure” on page 27

This chapter describes topology and installation options to provide an optimal execution environment for elastic and high performance Java Platform, Enterprise Edition (JEE) applications.

- ▶ Chapter 4, “Capacity planning and tuning” on page 49

This chapter describes performance capacity planning and JVM tuning for system administrators.

- ▶ Chapter 5, “Grid configuration” on page 69

This chapter describes how to configure WebSphere eXtreme Scale as the data grid and plug-ins that affect the behavior of the grid.

- ▶ Chapter 6, “Performance planning for application developers” on page 109

This chapter describes performance considerations for application developers.

- ▶ Chapter 7, “Operations and monitoring” on page 137

This chapter describes how to manage a WebSphere eXtreme Scale environment in production. We assume that you already have a WebSphere eXtreme Scale topology in place, together with one or more applications.



WebSphere eXtreme Scale architecture

This chapter provides a quick overview of the architecture of a WebSphere eXtreme Scale grid. The terms and concepts introduced in this chapter are important to the understanding of subsequent chapters.

This chapter includes the following topics:

- ▶ A grid from a user's point of view
- ▶ Grid component overview
- ▶ A grid from WebSphere eXtreme Scale's view
- ▶ Client grid access
- ▶ WebSphere eXtreme Scale internal components

2.1 A grid from a user's point of view

Figure 2-1 gives a logical user view of a grid. A user connects to a grid, then accesses the maps in that grid. Data is stored as key value pairs in the maps. Figure 2-1 shows grid A, which has two maps: A and B.

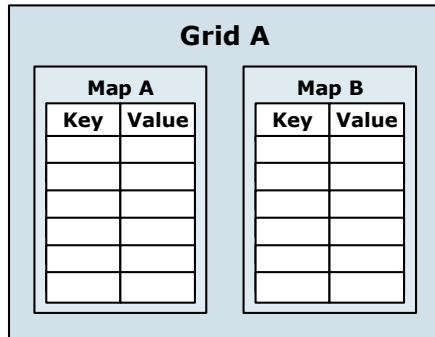


Figure 2-1 User view of a WebSphere eXtreme Scale grid

The term “grid” is often used loosely to refer to the entire WebSphere eXtreme Scale infrastructure, or that part that stores and manages your data. However, the term *grid* has a precise technical definition in WebSphere eXtreme Scale, namely, a container for maps that contain the grid’s data. Clients connect to grids, and access the data in the map sets they contain.

Figure 2-2 exposes the next level of detail by introducing the map set and partitioning concepts. A *map set* is a collection of, or container for, maps. So, a grid is really a collection of map sets. The key abstraction introduced by a map set is that it can be partitioned and otherwise configured in a unique way. Therefore, it can be split into parts that can be spread over multiple containers.

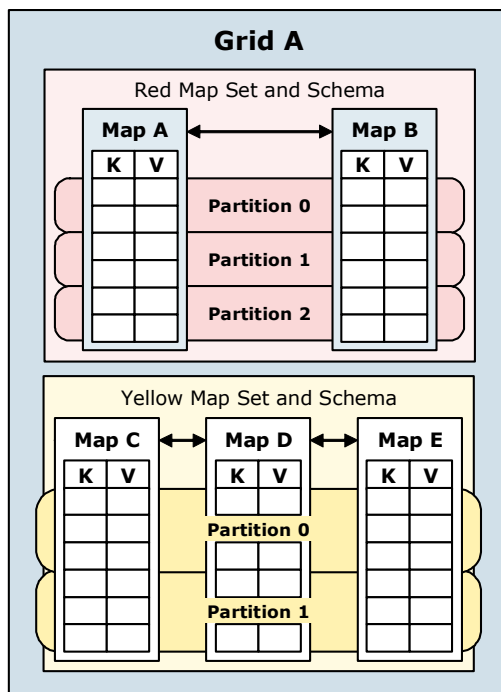


Figure 2-2 Maps in partitioned map sets in a grid

2.2 Grid component overview

Now, we provide more detailed definitions of our terms:

- Map

A map is an interface that stores data as key-value pairs (duplicate keys are not supported). A map is considered an associative data structure, because it associates an object with a key. The key is the primary means of access to the data in the grid. The value part of the map is the data typically stored as a Java object. A WebSphere eXtreme Scale map is conceptually similar in certain ways to the familiar `java.util.Map` but has far more capability.

- Key

A key is a Java object instance that identifies a single cache value. Keys can never change and must implement the `equals()` and `hashCode()` methods.

- Value

A value is a Java object instance that contains the cache data. The value is identified by the key and can be of any type.

- Map set

A map set is a collection of maps whose entries are logically related. More formally, a map set is a collection of maps that share a common partitioning scheme. Key elements of this scheme are the number of partitions, and the number of synchronous and asynchronous replicas.

- Grid

A grid is a collection of map sets.

- Partitions

Partitioning is the concept of splitting data into smaller sections. Partitioning allows the grid to store more data than can be accommodated in a single Java virtual machine (JVM). It is the fundamental concept that allows linear scaling. Partitioning happens at the map set level. The number of partitions is specified when the map set is defined. Data in the maps is striped across the N partitions using the key object's `hashCode()` method (or the method in `PartitionableKey` if your key implements this interface) modulo N . Choosing the number of partitions for your data is an important consideration when configuring and designing for a scalable infrastructure.

Figure 2-2 on page 14 shows two map sets in a grid: Red and Yellow. The Red map set has three partitions, and the Yellow map set has two partitions.

- Shards

Partitions are logical concepts. The data in a partition is physically stored in shards. A partition always has a primary shard. If replicas are defined for a map set, each partition will also have that number of replica shards. Replicas provide high availability for the data in the grid.

- Grid containers

Grid containers host shards and thus provide the physical storage for the grid data.

- Grid container server

A grid container server hosts grid containers. It is WebSphere eXtreme Scale code running either in an application server or a stand-alone JSE JVM. Often, grid container servers are referred to simply as JVMs.

Figure 2-3 on page 16 shows how these concepts are related.

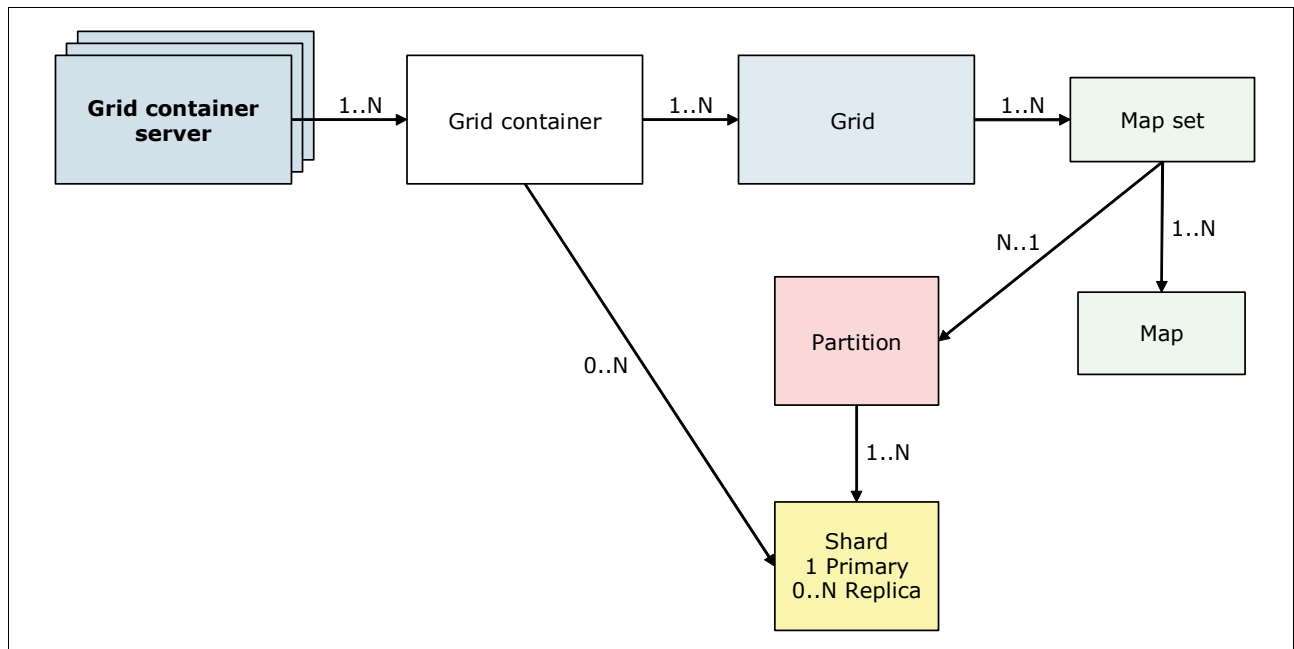


Figure 2-3 WebSphere eXtreme Scale metamodel

The following considerations apply:

- ▶ A grid container server can host many grid containers although it is common for it to host only one.
- ▶ A grid container hosts shards, which might be from one or more grids. A grid can be spread across many grid containers. The catalog service places shards in grid containers.
- ▶ A grid consists of a number of map sets. A map set is partitioned using the keys of the maps in that grid. Each map in the map set is defined by a BackingMap.
- ▶ A map set is a collection of maps that are typically used together. Many map sets can exist in one grid.
- ▶ A map holds (grid) data as key value pairs. It is not required but is common practice that a map uses only one Java type for keys and one other type for values; this practice makes the design and use of the map simpler.
- ▶ A map set consists of a number of partitions. Each partition has, for each map in that map set, a primary shard and *N* replica shards.

The example that is shown in Figure 2-4 shows the parts and concepts that we have described.

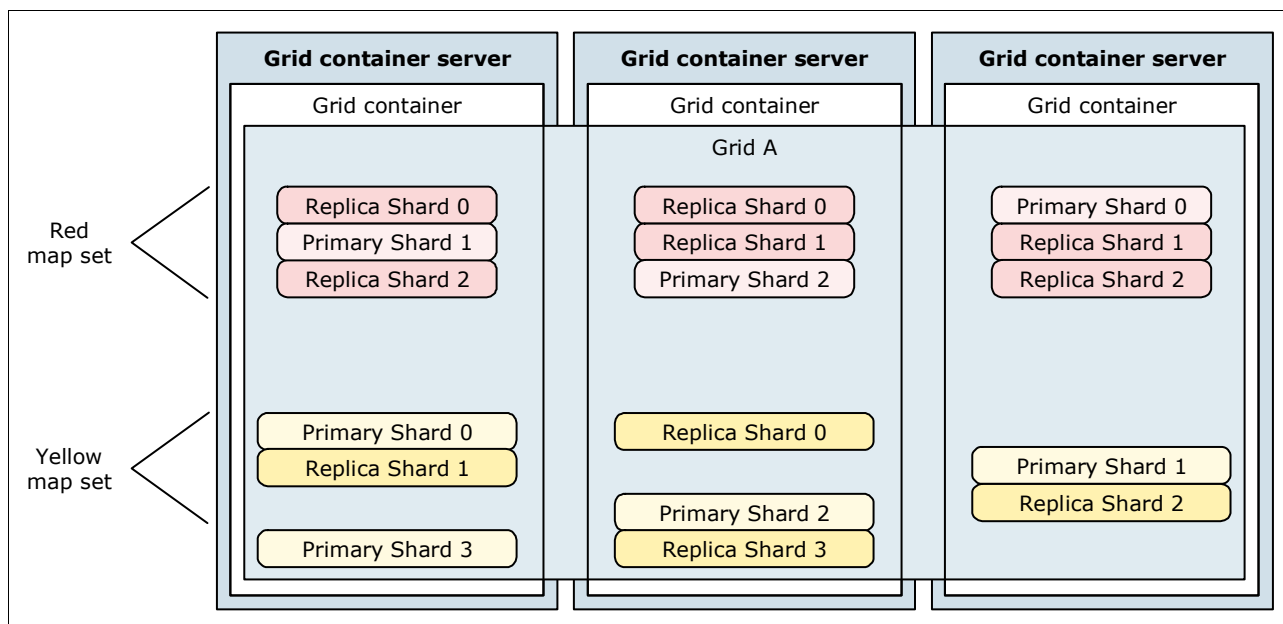


Figure 2-4 Grid A is stored physically in shards

Figure 2-4 shows an example of how grid A can be physically stored in WebSphere eXtreme Scale. Grid A contains two map sets: Red and Yellow. The Red map set has three partitions and two replicas, for a total of nine shards (nine comes from three partitions times three shards for each partition, the primary plus two replicas). The Yellow map set has four partitions and one replica, for a total of eight shards. Figure 2-4 shows one way that the shards can be distributed over the available grid containers. Of course, in a real installation, typically, there are many more partitions, but using three servers and one replica is quite realistic.

2.3 A grid from WebSphere eXtreme Scale's view

In WebSphere eXtreme Scale, there are two kinds of servers: catalog servers and (grid) container servers. *Catalog servers* are the JVMs that comprise the catalog service. *Container servers* hold the shards, which make up the data contained in the whole grid.

2.3.1 Catalog service

The catalog service maintains the healthy operation of grid servers and containers. The catalog service can run in a single catalog server or can include multiple catalog servers to form a highly available *catalog service domain*. The catalog service becomes the central nervous system of the grid operation by providing the following essential operation services:

- ▶ Location service to all grid clients (what partitions are where)
- ▶ Health management of the catalog and grid container servers
- ▶ Shard distribution and redistribution
- ▶ Policy and rule enforcement

A catalog server uses a host name and two ports to communicate with its peer catalog servers to provide the catalog service for the grid. This set of host names and port numbers from all the catalog servers defines the catalog service, and it is the grid's catalog service endpoints. Each catalog server must be given the complete set of catalog service endpoints when it starts. The catalog servers communicate among themselves on these ports to provide a highly available catalog service for the grid. Each catalog server listens on a separate port for container servers and grid clients. This combination of a host name and port on which a catalog server listens is also called a *catalog server endpoint*. You have to keep the context straight. Usually, you deal with container servers and clients, so this use is the predominant sense of the term. When a container server starts, it is given its catalog server endpoint, by which it will connect to the catalog service.

The good news is that a WebSphere eXtreme Scale grid is defined by the catalog service's catalog service endpoints. Container servers and clients connect to the grid by using a catalog server's client catalog service endpoint.

Container servers connect to the catalog service when they start. The catalog service thus knows the identity of all its container servers, and it distributes the shards over these containers. Catalog servers are not involved in normal gets and puts to the grid, so they are not a bottleneck that interferes with scaling.

2.3.2 Shards

The catalog service plays an instrumental role in the elastic nature of the grid configuration. It is responsible for the replication, distribution, and assignment of the shards (containing the data) to grid containers, as illustrated in Figure 2-5 on page 19. The catalog service evenly distributes the primary shards and their replicas among the registered container servers.

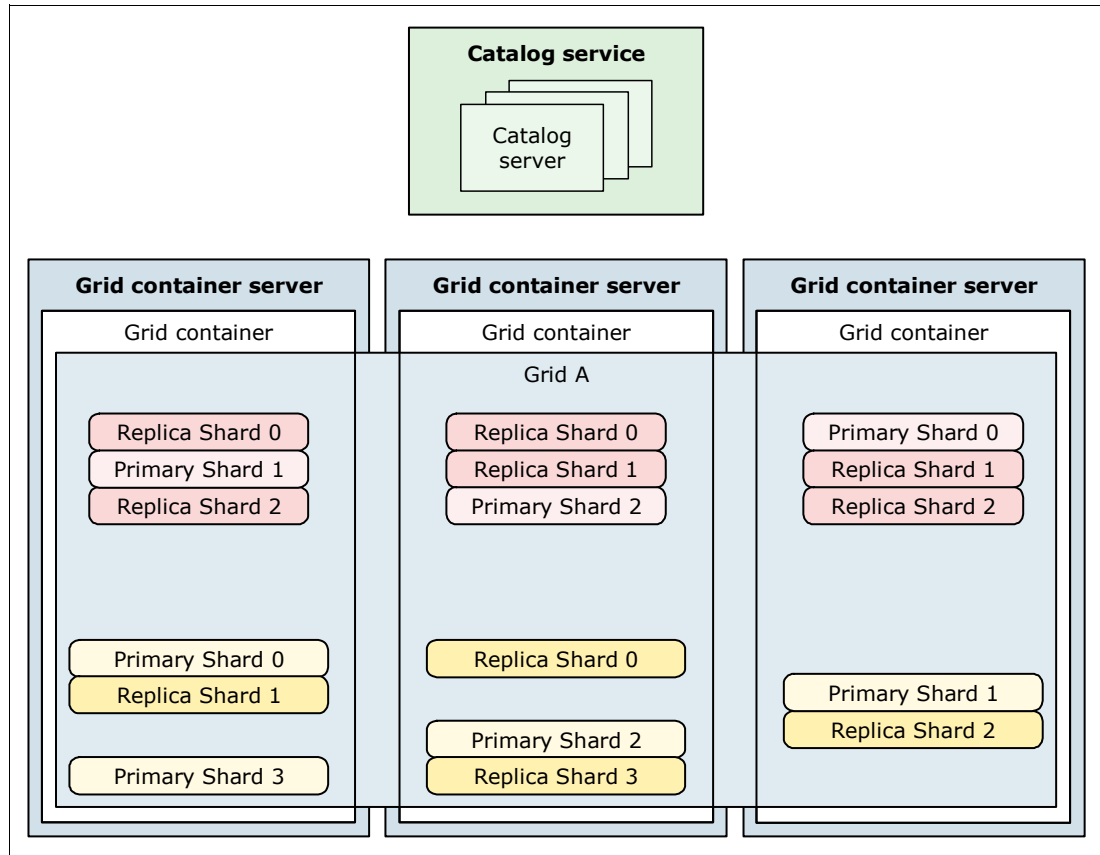


Figure 2-5 WebSphere eXtreme Scale grid

The shard distribution algorithms ensure that the primary and replica shards are never in the same container server (more specifically, never in two containers that have the same IP address) to ensure fault tolerance and high availability.

If the machine or grid container hosting the primary shard fails, the catalog service promotes a replica shard to be the primary shard, and creates a new replica shard in another grid container on another IP address (usually a separate machine). The replica shard is then populated in a background thread by copying the data from the new primary. If a machine or grid container hosting a replica shard fails, the catalog service creates a new replica in another grid container on another IP address, and is then populated by copying the data from the primary.

Shard types

There are three types of shards:

- Primary:
 - Handles read requests and all insert, update, and remove requests. (Replicas are read-only.)
 - Replicates data (and changes) to the replicas.
 - Manages commits and rollbacks of transactions.
 - Interacts with a back-end data store for read and write requests.
- Synchronous replica:
 - Maintains the exact state as the primary shard.

- Receives updates from the primary shard as part of the transaction commit to ensure consistency.
- Can be promoted to be the primary in the event the primary fails.
- Can handle get and query requests if configured to do so.
- ▶ Asynchronous replica:
 - Might not have exactly the same state as the primary shard.
 - Receives updates from the primary after the transaction commits. The primary does not wait for the asynchronous replica to commit.
 - Can be promoted to be a synchronous replica or primary in the event of the primary or a synchronous replica failure.
 - Can handle get and query requests if configured to do so.

Primary shards: A primary shard is sometimes referred to as the *primary partition*. While you might see those two terms used interchangeably, a partition is composed of a primary shard and zero or more replica shards, and includes shard data for all maps in that map set.

As the grid membership changes and new container servers are added to accommodate growth, the catalog service pulls shards from relatively overloaded containers and moves them to a new empty container. With this behavior, the grid can scale out, by simply adding additional grid container servers. Conversely, when the grid membership changes due to the failure or planned removal of grid container servers, the catalog service will attempt to redistribute the shards that best fit the available grid container servers. In such a case, the grid is said to scale in. Scaling out and in is accomplished with no interruption to application use of the grid (except perhaps a slight temporary slowdown). The ability of WebSphere eXtreme Scale to scale in and scale out provides tremendous flexibility to the changing nature of infrastructure.

Shard placement: Water flow algorithm

The mechanism employed to distribute shards among the available grid containers is based on an algorithm resembling the natural flow of water. Just as water will spread evenly but at changing depth over a shallow pan as the size of the pan grows or shrinks (this example is a hypothetical “rubber pan”), as grid container servers leave and join the grid, shards are redistributed.

This redistribution maintains shard placement rules, such as not placing primary and replica shards in the same container (or even the same machine to maintain high availability). 3.3, “Zone-based topology” on page 36 introduces another important shard placement rule: that primary and replica shards must not be placed in the same zone.

It is important to understand the implications of the shard placement policy defined and enforced by the catalog service. The water flow algorithm ensures the equitable distribution of the total number of shards across the total number of available containers. Hence, WebSphere eXtreme Scale ensures that no one container is overloaded when other containers are available to host shards.

WebSphere eXtreme Scale also enables fault tolerance when the primary shard disappears (due to container failure or crash) by promoting a replica shard to be the primary shard. If a replica shard disappears, another is created and placed on an appropriate container, preserving the placement rules. Other key aspects of the approach are that it minimizes the number of shards moved as well as the time required to calculate the shard distribution changes. Both of these concerns are important to allowing linear scaling.

Partitioning (assigning data to a particular partition) is of equal importance in ensuring that no one container is overloaded. Partitioning is controlled by the hash (or PartitionableKey) algorithm in your key class so it is important that this algorithm yield an even distribution. The default Java hash algorithm generally works fine, but you can write custom code if necessary.

To ensure high (or continuous) availability of a data partition, WebSphere eXtreme Scale ensures that the primary and replica shards of a partition are never placed in the same container or even on the same machine.

Figure 2-6 shows four partitions, each with a primary shard and one replica shard, distributed across four containers.

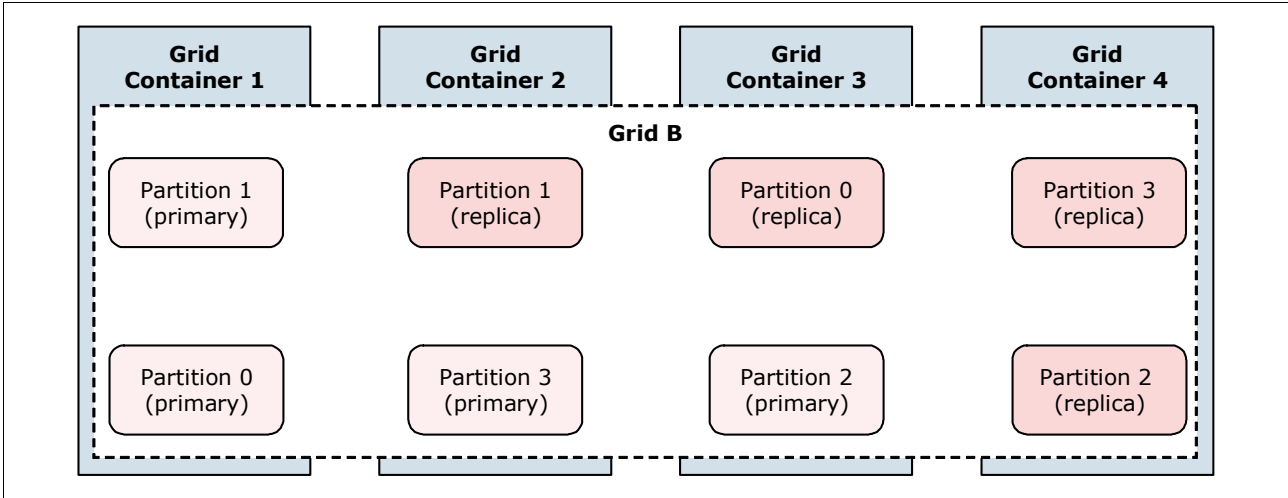


Figure 2-6 Shards placed on all available containers

Figure 2-7 shows how the shard placement adjusts if one of the containers failed and only three containers were available for placement. In this case, no primary partitions were affected so no replicas were promoted to be primary partitions. The two failed replica partitions are simply recreated in another container in the grid. You can turn off this automatic replacement if you want.

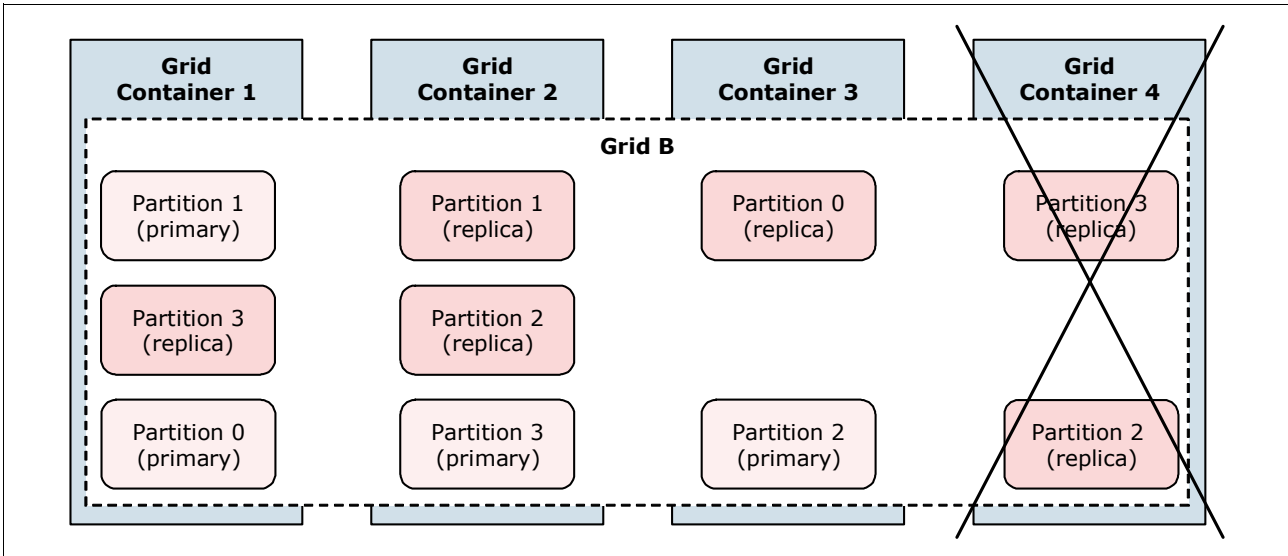


Figure 2-7 Shard placement and redistribution after Grid Container 4 JVM failure

2.4 Client grid access

Users access data in a grid through the WebSphere eXtreme Scale grid client, which is WebSphere eXtreme Scale code packaged in an ogclient JAR file. This JAR file requires no license for use. The primary user input is a catalog service endpoint, and grid and map names. When we describe the grid client in this section, we refer to the function that this WebSphere eXtreme Scale code performs. The user writes no such code.

The grid client begins its access to the grid by obtaining a routing table from the catalog service. Given an object to access, the client calculates the key's hash code, divides that by the number of partitions, and the remainder is the number of the partition containing the object. The routing table enables the client to locate the partition's primary shard, which contains the desired object. In the event of a container failure, or redistribution of partitions due to a change in grid container membership, the client will obtain an up-to-date routing table from one of the grid catalog servers when the next client request fails due to incorrect routing table content.

When a client is unable to get a response from any the containers hosting a shard from its routing table, the client will contact the catalog service again. If the catalog service is not available, the client fails.

Clients have minimal contact with the catalog service after the routing table is obtained, and network traffic is minimized when shards move, both of which enhance linear scalability.

2.5 WebSphere eXtreme Scale internal components

The following sections define the components in WebSphere eXtreme Scale. Figure 2-8 is an illustration of the terms that are described.

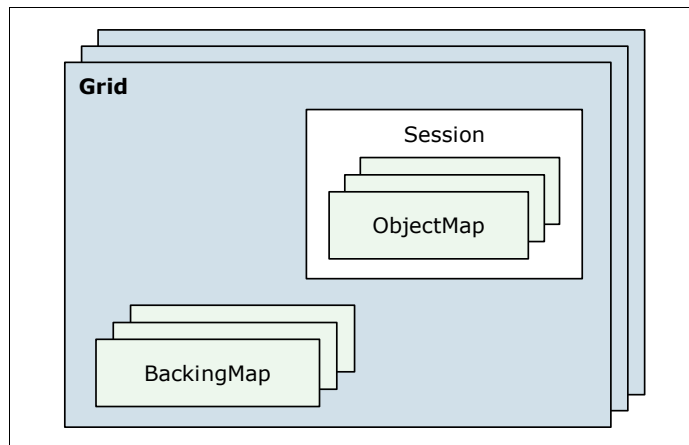


Figure 2-8 WebSphere eXtreme Scale internal components

2.5.1 Session

When a user application initially interacts with the ObjectGrid object that it looked up by name, it must first establish a session by getting a Session object from this grid object. Applications begin and end transactions using the session interface, or a transaction will begin and end

automatically. Because the session includes a transaction, sessions are not usually shared concurrently by multiple threads. Other threads (perhaps representing other users) can get their own sessions and have their own transactions.

2.5.2 Map

A *map* is an interface that stores data as key/value pairs. There are no duplicate keys in a map nor can there be null keys. There can be duplicate values and null values. A map is considered an associative data structure, because it associates an object (the value) with a key.

2.5.3 ObjectMap

An *ObjectMap* is a type of map that is used to store a value for a key. That value can be either an object instance or a tuple:

- ▶ An object instance requires its corresponding class file to be in the container server, because the bytecode is needed to resolve the object class.
- ▶ A tuple represents the attributes of an object. You do not need the class file present.

An *ObjectMap* is always located on a client, and it is used in the context of a local session.

Figure 2-9 illustrates three separate *ObjectMaps*. An *ObjectMap* holds key objects and value objects. From left to right, the first *ObjectMap* contains an integer key and a string value. The next *ObjectMap* contains a (hex) integer key and a compound value. The last *ObjectMap* contains a compound key and a compound value. The compound key and value can each be custom Java objects of your own (or someone else's) design, or they can be a tuple.

ObjectMap	
Key	Value
1	Daniel
2	Jennifer
...	...

ObjectMap	
Key	Value
A123B45C	John::Person Name: John Last name: Doe Birth date: 08.07.1999
D567E89F	Jane::Person Name: Jane Last name: Doe Birth date: 03.05.1979
...	...

ObjectMap	
Key	Value
John Key::Person Key Last name: Doe Birth date: 08.07.1999	John::Person Last name: Doe Birth date: 08.07.1999 Name: John Gender: Male
Jane Key::Person Key Last name: Doe Birth date: 03.05.1979	Jane::Person Last name: Doe Birth date: 03.05.1979 Name: Jane Gender: Female
...	...

Figure 2-9 *ObjectMap* examples

2.5.4 Tuples

A tuple is another way to represent compound objects. A *tuple* is simply an array of primitive types. It contains information about the attributes and associations of an entity. If you choose to use the WebSphere eXtreme Scale EntityManager, the EntityManager converts each entity object into a key tuple and a custom value tuple representing the entities (Figure 2-10 on page 24). This key/value pair is then stored in the entity's associated *ObjectMap*. If you use the WebSphere eXtreme Scale *ObjectMap* application programming interfaces (APIs), you can create key and value tuples yourself.

ObjectMap	
Key	Value
Tuple	Tuple
Tuple	Tuple
Tuple	Tuple

Figure 2-10 Tuples

The value of tuples is in not having to define custom classes when several objects (entities) are represented or packaged together. When a loader is used with the map, the loader will interact with the tuples.

Because a tuple is, for example, an ArrayList of Strings, Integers, Timestamps, and Booleans and all these types are defined in the Java Development Kit (JDK), there is no need to have a .jar file with custom classes installed on the container servers. If you wrote a custom MyObject class with those same Strings, Integers, and so on, a .jar including MyObject must be installed on the containers. For many installations, this arrangement is no problem at all and the use of custom Java classes is a common technique. If having a separate .jar file with your data classes and installing that .jar on both your client JVMs and the WebSphere eXtreme Scale container JVMs is a problem, the use of tuples might help you. The use of COPY_TO_BYTES is another way to avoid this problem (see 6.1.5, “COPY_TO_BYTES mode” on page 113).

2.5.5 Backing maps

A *backing map*, represented by a BackingMap object, contains cached objects that have been stored in the grid. An ObjectMap and a BackingMap are related through a grid session. The session interface is used to begin a transaction and to obtain an ObjectMap, which is required for performing transactional interactions between an application and a BackingMap object. There are also times, such as for agents or plug-ins that execute on a particular container or for a particular partition, that an ObjectMap represents the subset of a BackingMap’s data that belongs to that container or partition.

ObjectMaps and BackingMaps can reside in the same JVM that hosts your application; this design is the case for a local grid. BackingMaps can also reside in a container server that is separate from the ObjectMaps, and the two maps will communicate remotely to persist data.

Each key and value type pair, or each entity, if using EntityManager APIs, has its own BackingMap. Any serializable data (keys and values, or entities) are persisted to the BackingMap. Therefore, each BackingMap has its own loader class; furthermore, as shown in Figure 2-11 on page 25, each partition (each primary shard) has its own instance of that loader class, which can run in its own thread parallel to others. The BackingMap will request any needed data that it does not contain from its loader, which in turn, will retrieve it from the backing store. This process is illustrated in Figure 2-11 on page 25. Note that, as shown in the Figure 2-11 on page 25, loaders work on primary shards only, not replicas.

Loaders and the BackingMap: A *loader* is a plug-in to the BackingMap. The loader is invoked when data is requested from the grid and that data is not already in memory or when data is changed in the grid and the associated transaction is committed (or flushed). The loader has the logic necessary for reading and writing data to the backing store, which is applicable in a write-through and write-behind type of topology where the application's interactions are directly to the grid and it does not access the backing store itself. Built-in Java Persistence API (JPA) loaders are provided to simplify this interaction with relational database back ends. The JPA loaders use JPAs to read and write the data to the database back ends. Users provide a `persistence.xml` file to plug in the JPA provider, such as OpenJPA or Hibernate. A service provider interface (the "Loader" interface) is also provided to support other back-end data stores, or if you prefer to avoid using JPA or Hibernate, to deal with the database using Java Database Connectivity (JDBC) and custom code.

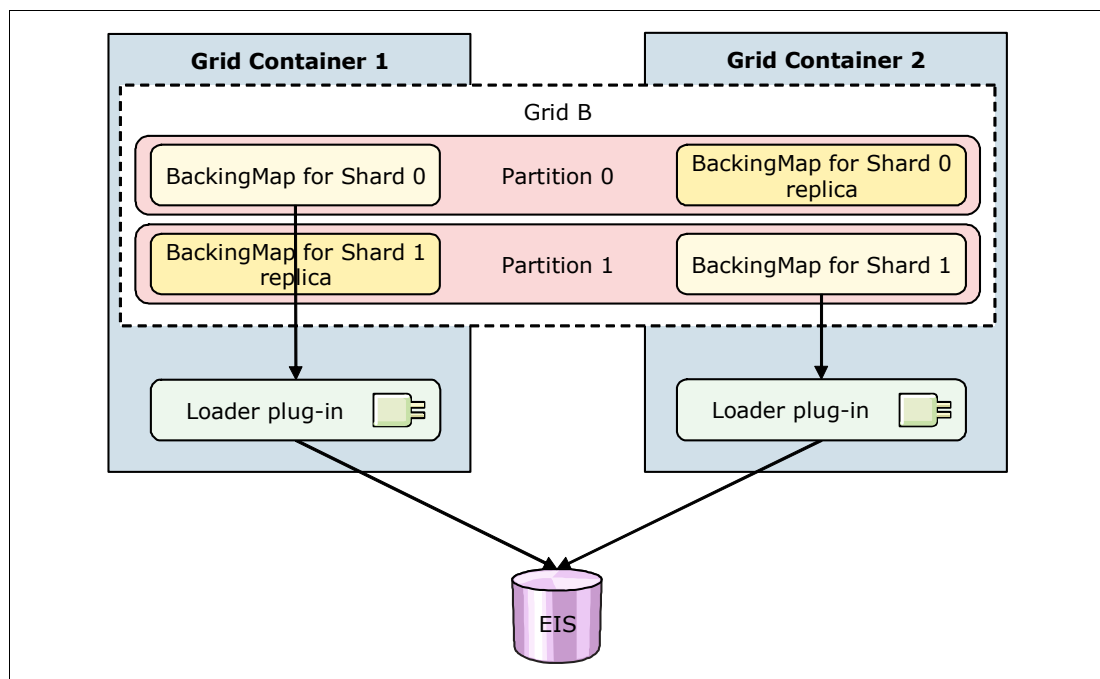


Figure 2-11 Example use of a BackingMap with a loader

2.5.6 Grid clients and backing maps

We use the following terms to describe how an application interacts with the grid:

- **Grid client**

A grid client is WebSphere eXtreme Scale code from the `ogclient.jar` file that interacts with grid servers on behalf of application code. The WebSphere eXtreme Scale APIs called by application code are all in the grid client jar file. Grid clients connect to a grid through the catalog service. Grid clients examine the key of the application data to route the request to the proper partition. A grid client contains a transaction-scoped **ObjectMap** and can contain a near-cache **ObjectMap** shared by all transactions for that grid client. Behind the **ObjectMaps** is the **BackingMap** (local or remote).

A grid client accesses data in a given named map through a hierarchy of layers, accessed in this order:

- a. A transaction-scoped ObjectMap containing only data that it has accessed during the current transaction (via the current WebSphere eXtreme Scale session).
- b. A near cache map (present by default but which can be disabled) that is seen by all transactions through all sessions in this client JVM.
- c. The BackingMap that is seen by all transactions through all sessions on all client JVMs. The BackingMap logically contains all data in the named map. The data is partitioned across all container JVMs, and there is a separate internal thread that handles each partition.
- d. The database or other enterprise information system (EIS) accessed with a loader plug-in attached to this BackingMap. A separate instance of the Loader class executes for each partition.

Clients can read data from multiple partitions in a single transaction. However, clients can only update data in a single partition in a transaction. A client can update multiple objects in multiple BackingMaps as long as the data has all been assigned to the same partition. This assignment might be done by accident or more commonly by application design using a small bit of custom hash code or PartitionableKey logic.

► ObjectGrid Instance

Applications must obtain an ObjectGrid instance to work with a grid. This way, the application can interact with the grid and perform various operations, such as create, retrieve, update, and delete the objects in the grid.



Topology and infrastructure

This chapter describes topology and installation options to provide an optimal execution environment for elastic and high performance Java Platform, Enterprise Edition (JEE) applications.

This chapter includes the following topics:

- ▶ Topology design options
- ▶ WebSphere eXtreme Scale distributed topology
- ▶ Zone-based topology
- ▶ Multi-master replication topology
- ▶ Port assignments and firewall rules

3.1 Topology design options

The Java platform application that will use the grid drives the WebSphere eXtreme Scale infrastructure design. The technical considerations of WebSphere eXtreme Scale topology choices are based on the non-functional requirements of the Java platform application, such as performance, stability, and availability.

The WebSphere eXtreme Scale topology is essential to the success of the Java platform application. A well-planned WebSphere eXtreme Scale topology provides a foundation upon which you can construct a quality environment. A poorly planned WebSphere eXtreme Scale topology might force you to change the system design late in the engineering process, or worse, when the WebSphere eXtreme Scale system is already in production.

In this section, we define what we mean by *WebSphere eXtreme Scale topology*. We explain our objectives in describing topology options, and we review common WebSphere eXtreme Scale topologies.

Catalog service domain: A *catalog service domain* (or simply *domain*) is one or more grids (set of containers) and their controlling catalog service. The concept of a domain is similar to a “cell” in WebSphere Application Server Network Deployment. Catalog service domains define a group of catalog servers that manage the placement of shards and monitor the health of container servers in your data grid.

3.1.1 Understanding WebSphere eXtreme Scale topology

In computing, *topology* refers to the layout or high-level configuration of a system, particularly an infrastructure system, such as a WebSphere Application Server cluster. Networking topology is another example of an infrastructure system. Networking topologies, such as a star or tree topology, define the layout of a network and the data flow among its components.

A topology can be either physical or logical. The physical aspect of topology defines the layout and the connectivity among the components of the system. The logical aspect of the topology demonstrates the communication or data flow among system components.

A WebSphere eXtreme Scale topology has the following attributes:

- ▶ It defines the layout of the WebSphere eXtreme Scale system, for example, the locations of clients, JVM instances, and WebSphere eXtreme Scale servers.
- ▶ It demonstrates the relationship and data flow between WebSphere eXtreme Scale system components.
- ▶ It typically includes a system diagram that shows the layout and high-level configuration of the WebSphere eXtreme Scale deployment.
- ▶ It usually includes a concise text document that explains the characteristics of the topology and the design objectives and rationale of the topology selection.
- ▶ It also includes certain configuration parameters that help define the topology, such as the number of partitions and number of replicas.

When getting started with WebSphere eXtreme Scale, try to make the topology as simple as possible. A simple topology makes it easier to understand situations that might arise, allowing you to debug one container server instead of multiple servers in a complex environment. Start with a simple topology, such as one catalog, one container, one partition, no replicas, and development mode set to true. If things go wrong, it is easier to track mistakes.

3.1.2 Stand-alone vs. managed installations

WebSphere eXtreme Scale can run as a stand-alone JVM or can be installed on WebSphere Application Server. Regardless of which type of server is chosen, the core capability of WebSphere eXtreme Scale, which is a transactional, secure, and scalable application cache fabric, is available to be exploited.

There are advantages to both approaches, and the issues to consider when deciding on which approach to use are environment-specific and a personal choice. Each topology that is described in this chapter can be implemented on either type of server or using a combination of servers of both types.

Managed grid

You can use WebSphere Application Server Network Deployment servers to host your WebSphere eXtreme Scale container servers. We refer to this design as a *managed grid*. The main benefit of this configuration is that you can manage your environment using the administrative capabilities that are available in the WebSphere Application Server Network Deployment product:

- ▶ The clustering and high availability management features offered by the managed environment can be exploited.
- ▶ Grid extensibility becomes relatively easier in a managed environment, because creating grid servers to extend the grid is only a matter of a few clicks (as long as the capacity supports the grid expansion).
- ▶ Configuring zones in a WebSphere Application Server Network Deployment environment is easily done using the node group capabilities.
- ▶ Additionally, the commonly available monitoring tools that might already be employed to monitor the performance and availability of your environment can be used to monitor the grid servers.

There is a cost for having access to these advanced features. One is in terms of purchasing the required WebSphere Application Server licenses. There also is likely to be a modest performance cost with this approach due to the overhead that is associated with the advanced application server features. This performance cost might be higher in cases where WebSphere Application Server client certificate authentication has been enabled.

Leading practice for performance in a managed grid:

WebSphere eXtreme Scale uses the object request broker (ORB) to communicate over a TCP stack. Using the proper configuration for the ORB properties that are used to modify the transport behavior of the data grid is essential for optimum performance.

For information about configuring the ORB in WebSphere Application Server and for recommendations on the settings, see these resources:

- ORB properties

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extremescale.admin.doc/rxsorbproperties.html>

- orb.properties tuning

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extremescale.prog.doc/cxsjvmtune.html>

Specifically, pay attention to the ORB thread pool (properties beginning with `com.ibm.CORBA.ThreadPool.`). This number might need to be increased.

Stand-alone grid

Stand-alone servers can use the Java Standard Edition (JSE) implementation of your choice. The main benefit of this configuration is the use of a less expensive JSE environment for grid containers and faster overall performance. If your environment already uses JSE for its applications, you might also be inclined to use JSE stand-alone containers. In this case, the inclusion of WebSphere eXtreme Scale as a platform might be the only new addition.

A common scenario is the use of a Network Deployment-managed environment to host enterprise applications and the use of a readily available JSE runtime environment as the grid layer.

While there might be cost advantages to using stand-alone servers, the downside is the lack of availability management and monitoring solutions for the JSE containers that host the grid servers. You will not have the administrative control that is provided with WebSphere Application Server, and certain monitoring techniques are not supported in a stand-alone environment. Tivoli® Performance Viewer and Cache Monitor run only in a WebSphere Application Server environment.

3.1.3 Local, embedded, and distributed grid topologies

In its simplest form, it is possible to create and use a WebSphere eXtreme Scale grid to cache data locally in any Java process. The application runs in the same JVM as the grid (referred to as an *embedded* or *collocated* grid) and is used as a local cache. The application can use transactions to write all or none (never partial) of a set of changes, an ability not possible with `java.util.Map` as your cache. The application can perform better by retrieving data from the cache rather than from calls to a database, but this topology does not provide for fault tolerance or high availability. When the application is deployed to a cluster of servers, each application instance accesses only the cache in its JVM. There is no replication among caches to keep the data in synch. In addition, the grid resources compete with application resources because they are in the same JVM.

Taking this topology a step further, you can distribute the grid across the application JVMs. The application instances are collocated with the grid, but the data is now spread across JVMs and might no longer be local to the application. This design introduces a measure of

fault tolerance and high availability by putting replica shards in a JVM other than the primary. However, you still have the issue of resource (mainly memory) allocation between the grid and application in the same JVM.

A more typical case is a grid that is distributed among many servers, separate from the application JVM. This design provides fault tolerance and high availability features, as well as the separation of JVM resources between the application and the grid. We describe this topology more in 3.2, “WebSphere eXtreme Scale distributed topology” on page 32.

3.1.4 Multiple data center topologies: Zone vs. multi-master replication

WebSphere eXtreme Scale is typically deployed for large, complex, and critical applications, for example, an online banking system for a multi-national financial services company. Usually, multiple data centers run these applications in order to provide high performance as well as failover and disaster recovery. The WebSphere eXtreme Scale zone-based topology and the multi-master topology (new in WebSphere eXtreme Scale V7.1) are good candidates for multiple data center deployment. It is common to use zones within a data center and multi-master between data centers.

CAP theorem: The CAP theorem states that a distributed computer system cannot support more than two of the following three properties: consistency (of data across nodes), availability, and partition tolerance. With regard to this theorem, the multi-master topology is considered to be an availability and partition-tolerant (AP) topology. The zone topology is considered a consistency and availability (CP) topology. The third combination consistency and availability (CA) does not scale and is not appropriate for an extreme transaction processing environment.

Zone-based topology

A zone-based topology spreads a single grid across multiple data centers through the use of zones. With zone support, the metadata shared by the catalog servers is synchronously replicated, while the data objects are copied asynchronously across networks. There is a single version of the data in the grid with no question as to the real value of the data. In the event that a data center is lost from the grid, replicas of the data can be promoted to be the primary data.

WebSphere eXtreme Scale zone-based topology is best for controlling the placement of primary and replica shards within a single domain. All servers in a single domain must be connected together with a speed equivalent to a normal single physical site. If it is possible to achieve this design over long distances with suitable fast connections, it is a workable solution as well. The goal with zone-based topology is, *within a domain*, to avoid both a given primary shard and its replicas being somewhere depending on a single point of failure (SPOF), for example, a single physical server, a single blade frame, or a single power supply.

Identify all SPOFs in a domain and identify all servers and other components dependent on that SPOF; there must never be one component on which every other component in the domain depends. After you have identified these situations, consider moving the servers and other components that are dependent on a given SPOF into a WebSphere eXtreme Scale zone.

This move will insure that if a primary shard is placed in a JVM in one zone, the matching replica will be placed in another zone. In this way, a failure of any SPOF will not cause the loss of both a primary and its replicas.

Leading practice: Never run zones across separate data centers unless you have special fast connections that are equivalent to speeds within a single data center (as is fairly common in Europe). If you do not have such connections, run separate grids and use multi-master replication instead.

Multi-master replication topology

A multi-master replication (MMR) topology links multiple grids across data centers. Each grid manages itself independently, and changes are replicated asynchronously to remote grids. MMR does not replicate on every transaction commit as zones do; instead it replicates less often with more data and thus makes more efficient use of communication capacity between geographically separate data centers. When changes are made to the same data in two separate grids, WebSphere eXtreme Scale can usually identify which change came last and thus must override the other change. For the cases where WebSphere eXtreme Scale cannot decide, a collision arbiter (usually a few lines of custom code that understand your data objects) is called during replication to determine how to resolve the difference.

Because MMR is new, the bulk of best use cases are still maturing and will accumulate in actual practices. It is not yet always clear which topology, zone-based replication or MMR, is best for a given situation (although communication speeds are often the deciding factor). There might well be no clear winner. Therefore, each practitioner must choose the best for the particular situation.

WebSphere eXtreme Scale multi-master replication is best for keeping multiple independent domains in “close-but-not-exact” synchronization. We say “close-but-not-exact synch” because multi-master replication by nature does asynchronous replication. As a result, there is always a window of difference (latency or staleness) between the domains or sites. In many situations, such latency is acceptable in exchange for the greater capacity over lesser bandwidth that MMR achieves.

If you cannot tolerate *any* latency between sites, you must obtain fast enough connections between your sites so that you can perform synchronous replication between zones. Because the size (in seconds or minutes) of the latency between MMR-linked sites is totally dependent on the wire speed between the sites and, to a lesser extent, CPU on the sites, you need to measure to see if your equipment achieves your goal. The measurements need to be taken as part of a valid performance test, which includes using a representative target load and realistic data.

It is a common practice to use both zones and MMR. Zones are used to eliminate any single point of failure within a data center and MMR is used to replicate between data centers. As with any other topology design, you must have a clear reason for using each of these features rather than using them because “everyone does”.

Caution: The current implementation of MMR will replicate all data in the grid on a cold start or cold restart. Plan your bandwidth between data centers accordingly if this approach will be an issue.

3.2 WebSphere eXtreme Scale distributed topology

A distributed WebSphere eXtreme scale topology, as shown in Figure 3-1 on page 33, represents the majority of use cases; even with multiple data centers, each data center is typically a distributed topology. In a distributed topology, the application servers host a grid client that communicates with grid servers to access data from far caches (“far” means in a

separate JVM, often across a wire). The data that is stored in the grid is spread across the JVM instances where WebSphere eXtreme Scale containers execute.

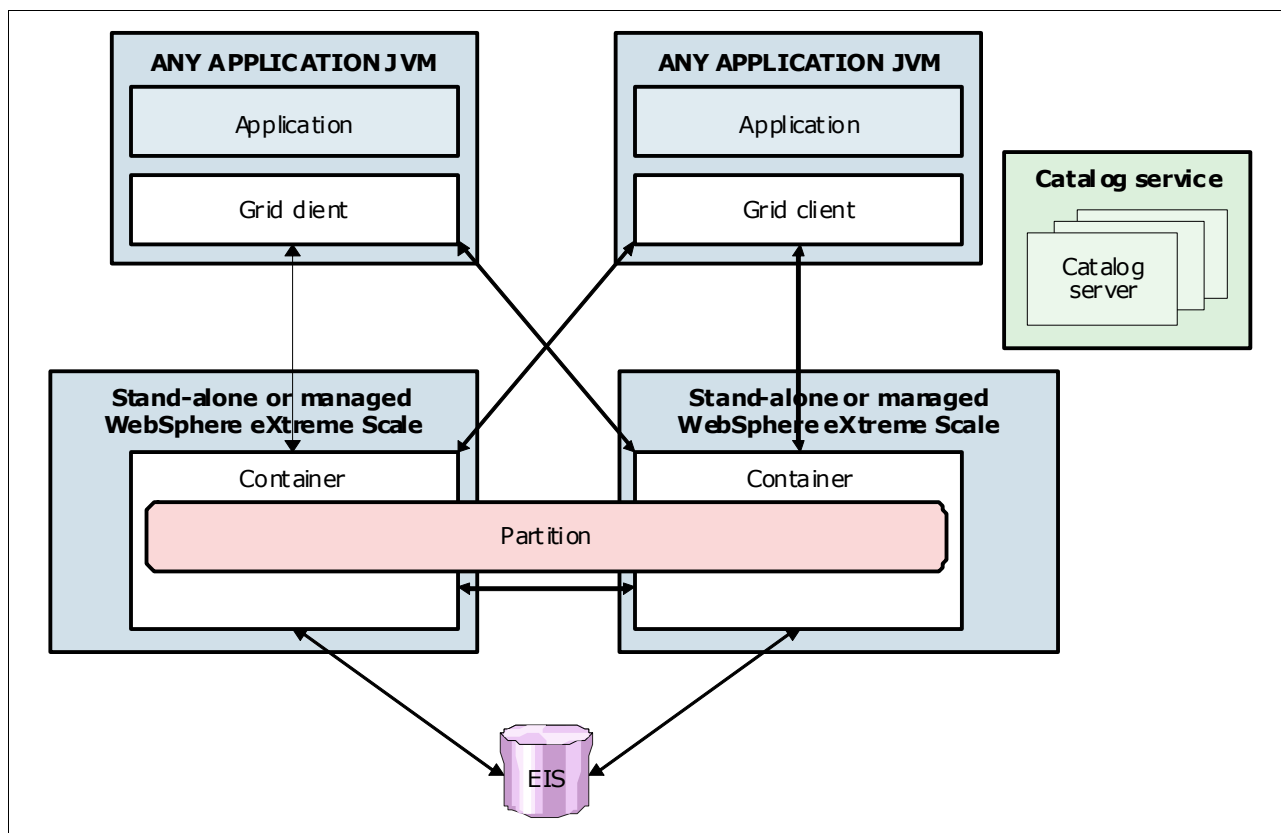


Figure 3-1 Distributed cache topology

A distributed topology is appropriate for most mission-critical enterprise web applications where managing heavy loads with large data, achieving linear caching performance, and executing large and complex application code are required. This topology provides fault tolerance and high availability features. The operation of the application and its server has no impact on the grid containers. The separation of JVM instances can help to avoid contention of system resources and ensure replication performance and system response time.

Caution: Be aware that problems might occur if you have catalog servers or container servers running in WebSphere Application Server processes with the same name. For example, hosting a catalog service in more than one node agent, or if you have containers running in servers with the same name on separate nodes (that is, server1). For more information, see 5.1.6, “Duplicate server names in WebSphere Application Server” on page 83.

3.2.1 Catalog service placement

The *catalog service* controls partition and shard placement and routing for all clients, making it a critical component of the grid. As such, the catalog service must be clustered for high availability and must be taken into account during the planning phase of the grid topology. Unlike the WebSphere Application Server deployment manager, ensure that at least one catalog server is up and functioning at all times; your grid is likely to become inaccessible if all catalog servers are down.

A catalog server can be configured and run in any JVM in the environment. A catalog server can be started as a stand-alone JVM or configured to be embedded into a WebSphere Application Server infrastructure process.

Consider installing the catalog servers on separate machines from the container servers. When these components are colocated on the same machines, they compete for the same resources, which can lead to problems. This design also ensures that maintenance can be applied to catalog servers without affecting the containers.

For high availability and redundancy, it is recommended that you run with three catalog servers in a production environment. More than three catalog servers is supported; however, it is not recommended to have a large set, such as ten. Typically, one catalog server per machine is a good reference.

The catalog servers communicate together and typically rely on maintaining a quorum to determine the changes that need to be made to the grid configuration. Unless overridden, quorum is the full set of catalog servers. The catalog service will only respond to container events while the catalog service has quorum. Using this mechanism ensures that the grid configuration is not corrupted in the event of a catalog server loss due to a JVM or network failure. Intentionally stopping a catalog server instance does not cause loss of quorum. When quorum is lost due to a temporary outage, it is re-established when the catalog server comes back online. When quorum is lost due to a more permanent or lengthy outage, the administrator can override quorum.

Catalog service in WebSphere Application Server

By default, the catalog service runs in the deployment manager process if the deployment manager node has WebSphere eXtreme Scale installed and the deployment manager profile is augmented. This approach is intended only for development or prototyping; for serious testing and production, do not use a catalog running in the deployment manager process. The catalog service can also run on any node agent or application server in the cell. The leading practice is to run the catalog service in a cluster of dedicated application servers so that no other processing competes with the catalog service in that JVM.

When running the catalog service in WebSphere Application Server, the service uses the WebSphere Application Server core group mechanisms. The members of the catalog service domain grid cannot span the boundaries of a core group, and a catalog grid therefore cannot span cells. However, WebSphere eXtreme Scale clients can span cells by connecting to a catalog server across cell boundaries, such as either a stand-alone catalog grid or a catalog grid that is embedded in another cell. Therefore, you can, if you want, have a cell containing only catalog and container servers and have it used by applications in other cells. You can even have catalog server clusters in one cell and various container server clusters in various other cells. Unlike catalog servers, container servers that are part of the same grid can be organized into more than one core group (to keep core group size smaller); each container server is in only one core group.

Leading practices

When designing your topology, consider the following leading practices:

- ▶ For availability, you must configure a catalog service domain. See *Best practice: Clustering the catalog service domain* at this website:
<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.e xtremescale.admin.doc/cxscatservbp.html>
- ▶ Designing high availability grids requires careful planning, sizing, and configuration. More details about high availability and failure modes can be found in the Availability topic of the

WebSphere eXtreme Scale Information Center. This topic is available at the following Web page:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.e xtremescale.over.doc/cxsavail.html>

- ▶ Do not collocate the catalog services with WebSphere eXtreme Scale container servers in a production environment. Include the catalog service in an application server cluster that is not hosting any other logic.

3.2.2 Example

Consider the following application:

- ▶ An application is deployed into three data centers, with two production data centers and one data center for disaster recovery.
- ▶ Each data center has two Network Deployment cells with a number of servers in each cell.
- ▶ There are two layers of WebSphere Application Servers in each data center:
 - The first layer of WebSphere Application Server has the UI application deployed. The UI application manages the user interface and authentication, authorization, and entitlement (AAE). It builds the right user interface (UI) for the right user with the right access to customer and customer relationship information. This UI application must access a back-end repository to determine an employee's AAE settings.
 - The second layer of WebSphere Application Server, BL, hosts the business logic of managing the customer relationship information. This application interacts with the main customer relationship repository back-end system. In addition, the second layer of WebSphere Application Server also hosts an application programming interface (API) that hundreds of the bank's applications call to interact with customer data.
- ▶ The application has a high volume during peak hours with 6000 requests per second with rigid response time and throughput requirements.

To improve the performance of the two layers of applications, we can deploy two WebSphere eXtreme Scale data grids, as shown in Figure 3-2 on page 36. Each data center looks like Figure 3-2 on page 36 and the data centers are linked (catalogs and containers) using MMR.

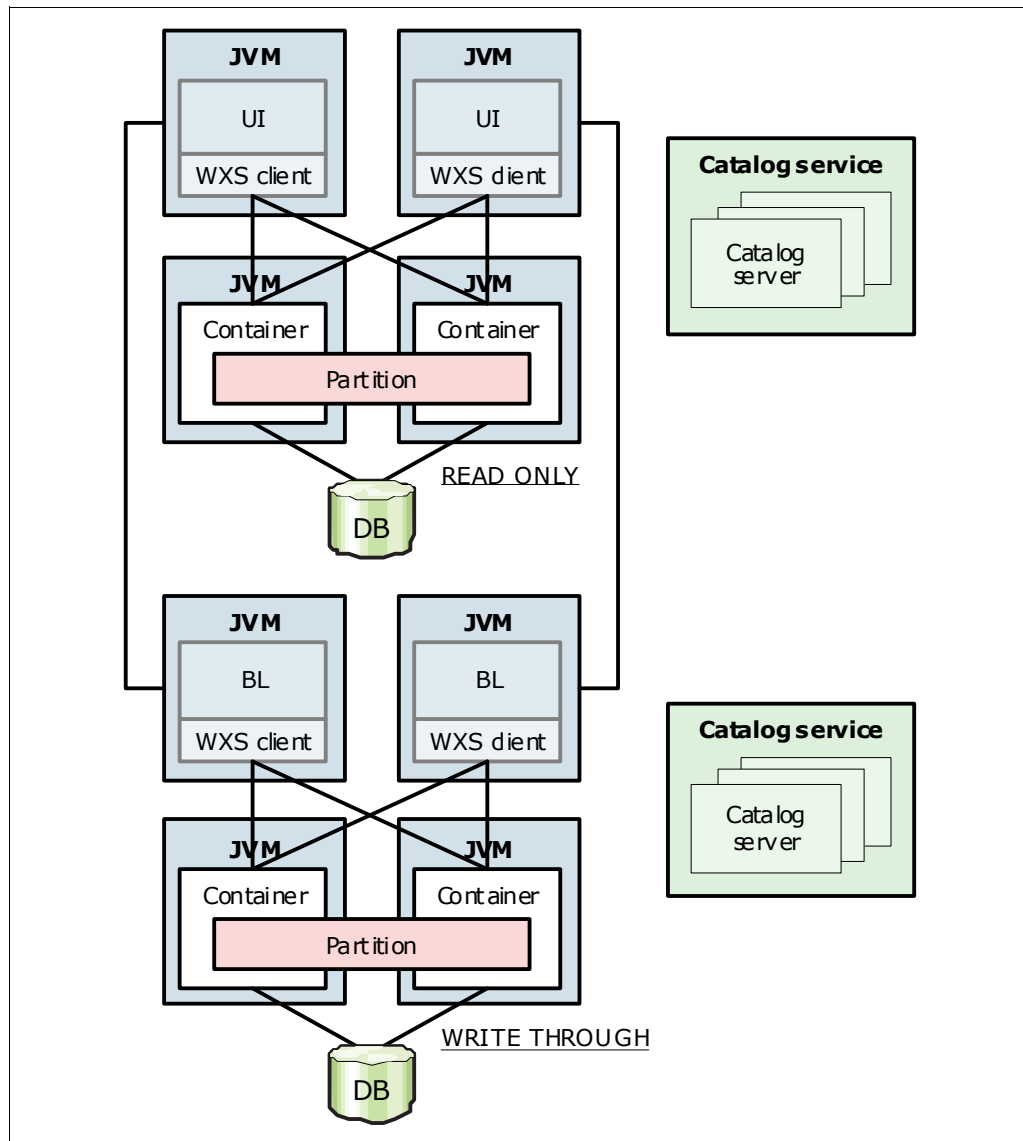


Figure 3-2 Sample distributed topology

The first data grid (the top pair of containers) resides on physical servers that are connected to the UI application servers. The grid works as an inline cache for the employee security data repository. The grid for the UI application is installed as a stand-alone WebSphere eXtreme Scale system, considering the relative data and operation simplicity.

The second layer of data grid is behind the business logic servers. Because the majority of the transactions are reading customer information, write-through caches are built. Primary and replica shards are distributed in a partition across the physical servers to provide better fault tolerance and high availability. The data grid servers reside in dedicated server hardware, providing maximum performance and ease of operation.

3.3 Zone-based topology

Mission-critical applications must be highly availability. An hour of unscheduled system downtime can result in large losses to the company and can have a long-term impact on

customer satisfaction. A zone-based topology can help you achieve high availability and failover capabilities within a single data center or (with sufficient communication speed) across geographical locations in a multiple data center environment.

A *zone* is a collection of grid containers administratively grouped for the purpose of better control over data placement. A zone-based topology (Figure 3-3 on page 37) is a special case of distributed WebSphere eXtreme Scale caching topology with primary and replica shards deployed across various zones in the same data center or in separate data centers. The server containers are controlled by a master catalog server and backup catalog servers deployed into separate zones. Multiple zones can be deployed across wide area networks (WANs) and local area networks (LANs). However, a particular zone is not designed to be deployed across WANs or into separate data centers. This limitation comes from the constraints of group service and the high availability manager.

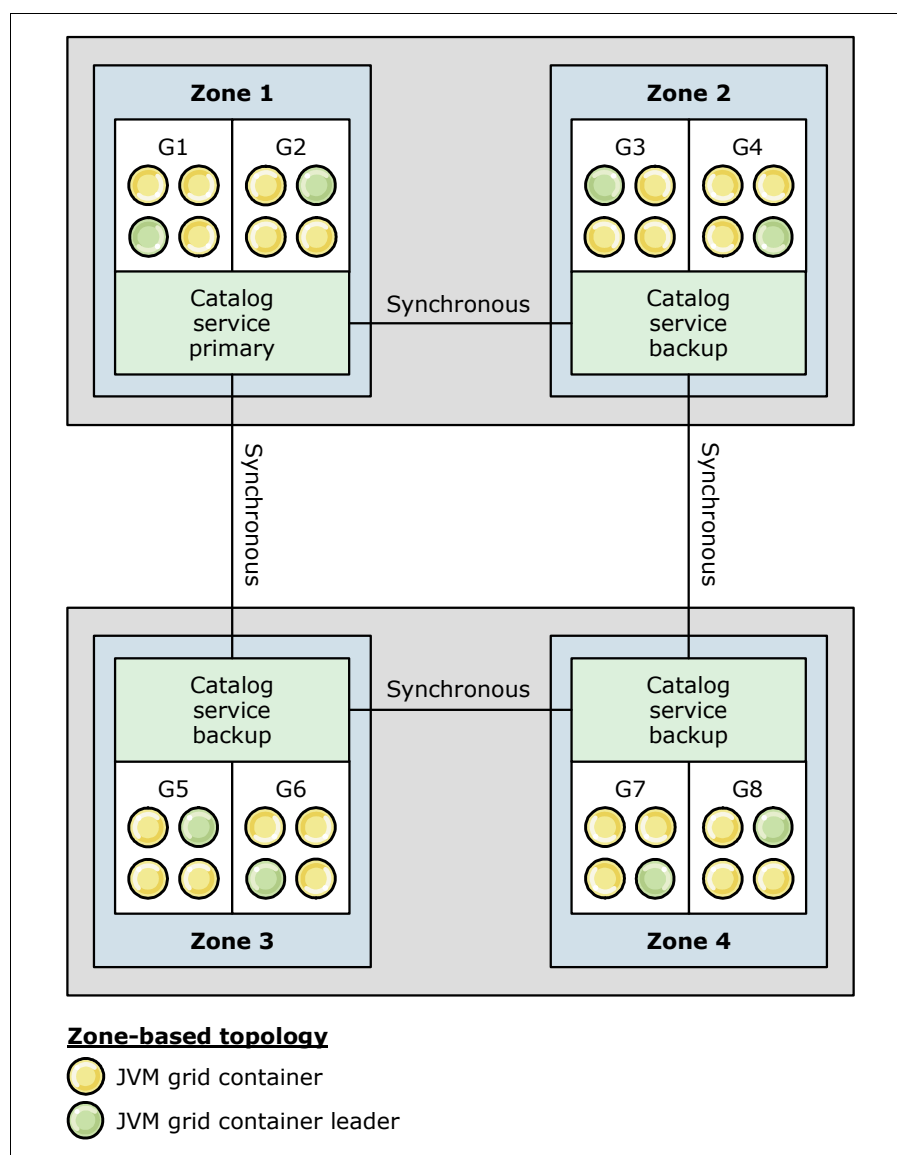


Figure 3-3 Zone-based topology

A zoned-based topology offers the following advantages:

- Data caches are deployed within the same data center or across data centers.

- ▶ Rule-based replication is performed.
- ▶ Primary and replica shards are deployed into separate geographical locations for high availability and disaster recovery.

3.3.1 When to use zone-based topology

For highly critical applications where both high performance and high availability are key requirements, a zone-based topology provides a best-fit caching strategy. This topology is particularly suitable for grouping servers to overcome a single point of failure scenario. The single point of failure does not have to be software or even hardware but can come from loss of electricity, loss of network service, or any event that impacts the physical location of the servers.

When considering a zone-based topology, the physical distance and the speed of network connectivity are practical design factors. A slow network and multiple single points of failure (which there usually are) can be a concern.

3.3.2 Synchronous and asynchronous replication

The primary and replica shards of a data grid are striped across zones. Zoning rules define the placement of synchronous or asynchronous replica shards in respective zones. The replication strategy ensures high availability within the data center and between data centers.

Synchronous replication within the zone takes advantage of the fast network within a data center, and therefore, is frequently the chosen mode of replication. Catalog servers are replicated via synchronization, but they replicate only the shared metadata.

If you find that you are not achieving your target response times for writes to the grid using synchronous replication, test with asynchronous replication. You might decide that the extremely brief latency between the primary shard and the asynchronous replica changes (which only matters on a container failure) is worth the improved performance.

3.3.3 The placement of catalog servers

With zones, you have one catalog service domain for the grid that spans the zones. Configure one catalog server per zone for high availability and failover purposes. If an object is not found within the local zone, the closest location of the data will be made available to the application.

3.3.4 Proximity-based routing

To reduce the traffic between zones, physical hosts, and processes, WebSphere eXtreme Scale clients need to set up a routing preference that favors proximity-based routing. *Proximity-based routing* supports routing preferences for resident zone, local host, and local process. This approach allows WebSphere eXtreme Scale to get the data that you request from the closest source; for example, it will use the catalog server in the same blade frame as your application or will (with `replicaReadEnabled=true`) get an entry from the shard (whether primary or replica) on the same floor as your application.

3.4 Multi-master replication topology

With a multi-master replication topology, you can replicate two or more grids across multiple data centers to create mirror images of the grids. Each data grid is hosted in an independent catalog service domain, with its own catalog service and container servers. The mirroring is accomplished using asynchronous replication between primary shards over links connecting the data grids together. A multi-master topology offers a balanced solution that addresses connectivity problems, disaster recovery, and service locality. The design of the topology determines the level of fault tolerance and availability. With multi-master replication, each domain can have its own primary copy of the data. Data updates to one primary are replicated asynchronously to linked primaries in a way that is especially efficient for data centers that are remote from each other. Data can be read locally from any data center.

Multi-master replication has limitations and is not appropriate in any situation where an availability and partition (AP) topology is not appropriate. For example, it might not be appropriate for applications with a database at each data center and with transactions that must commit to a database once and only once, such as applications that process purchases, accounts, or credit operations. If the other data centers are read only for these transactions, or share a single database, multi-master might be appropriate.

Multi-master replication has other limitations and dependencies:

- ▶ Domains must have access to all classes that are used as keys and values. Any dependencies must be reflected in all class paths for data grid container JVMs for all domains. If a CollisionArbiter plug-in retrieves the value for a cache entry, the classes for the values must be present for the domain that is starting the arbiter.
- ▶ Typically, preloading occurs when starting a data grid. However, multi-master topology does not require preloading of every data center. A catalog service domain automatically performs reloading from the contents of the linked domains so that only one domain needs to be preloaded.
- ▶ EntityManager is not supported, because a map set that contains an entity map does not get replicated among catalog service domains.
- ▶ Multi-master replication can only be configured for fixed partition grids (which is the default style of grid and is the most commonly used). Grids configured as “per container” cannot be replicated.

3.4.1 Topology options

There are several options for multi-master replication topologies. Figure 3-4 on page 40 shows three topology concepts:

- ▶ A serial chain of linked domains. The chain terminates with the last domain on each end of the chain. A break in the chain will isolate one or more domains.
- ▶ A ring of domains with a link between each domain. The domains are linked to form a full circle. Because each domain is linked to two other domains, a break in the circle will not stop replication.
- ▶ A hub and spoke configuration with a central hub that replicates with each domain. This topology is preferred over the chain or ring for large interconnected networks. Unlike the chain, it has only one point where failure can isolate domains, and unlike the ring, it does not replicate the same data multiple times.

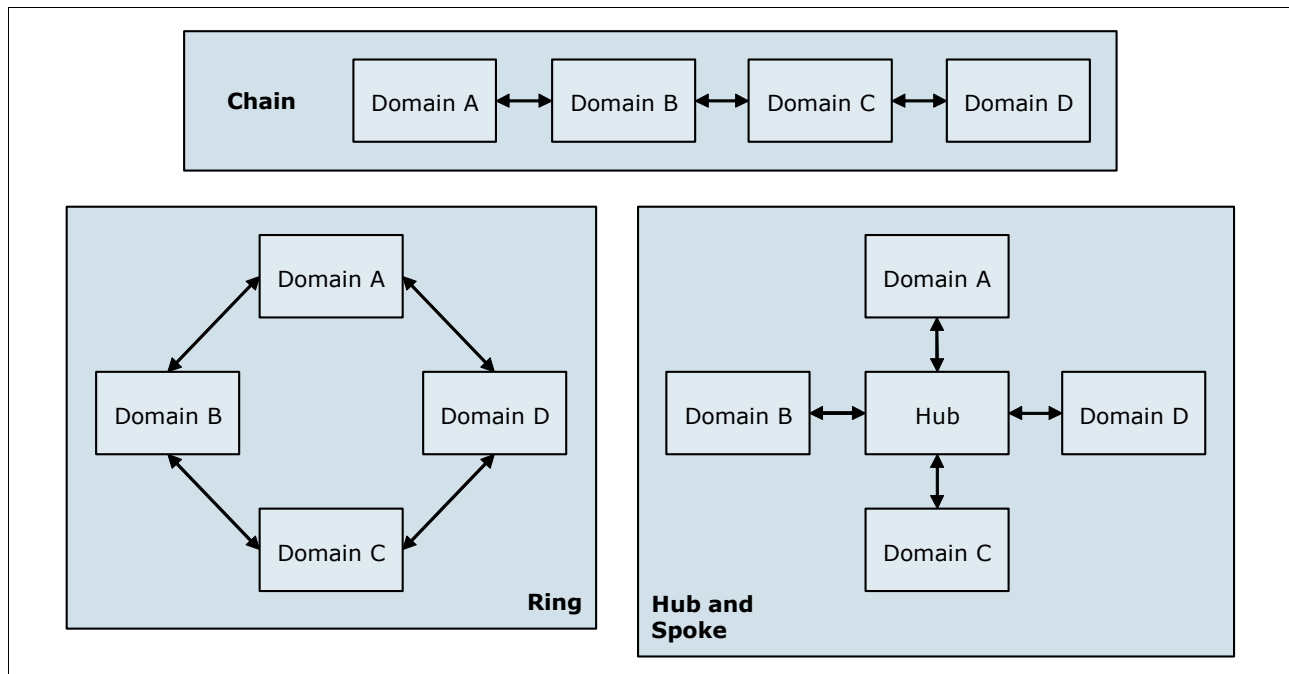


Figure 3-4 Multi-master replication topology options

In addition, there is the tree topology to consider. A tree topology, as shown in Figure 3-5, is similar to a hub and spoke topology in that each domain acts as a hub for its child domains. Because there is additional work that happens at a hub to resolve update collisions (see 3.4.3, “Collision management considerations” on page 41), this topology spreads this work across multiple (hub) domains and thus might be more suitable than a hub and spoke topology in installations with a large number of domains.

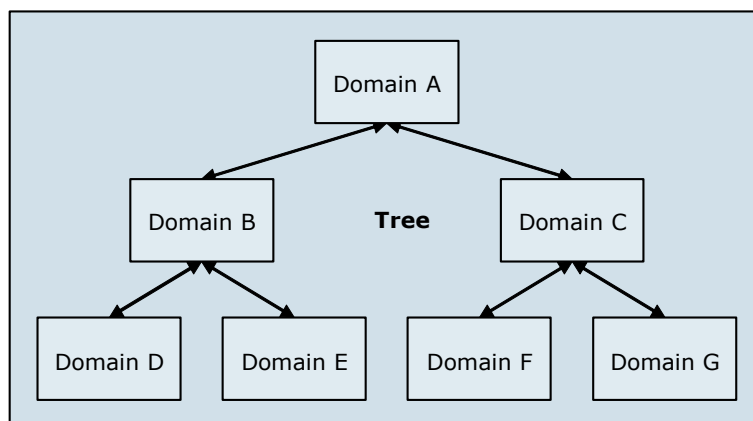


Figure 3-5 Multi-master replication tree topology

When designing a topology, the art is to engineer one that is as simple as possible for your situation. Do not use a complex topology or combine simple topologies into a highly complex one unless you must.

3.4.2 Performance and fault tolerance considerations

A WebSphere eXtreme Scale link establishes and manages a two-way asynchronous replication and update path between catalog services domains; more specifically, the path is

between matching BackingMap primary partitions in the domains. How you link the topology that you choose is central to the flexibility of multi-master replication. With this linking flexibility comes the task of designing a multi-master replication topology that balances among the replication performance, fault tolerance, and the speed of change replication.

Fault tolerance of the topology will depend on how the domains are linked. You can achieve higher fault tolerance by providing multiple links between domains. However, links consume system resources, such as network bandwidth. More links will have more system performance impact, so do not add links unless they are necessary. Remember that temporary link breaks that are quickly fixed might have no appreciable effect on replication, because multi-master replication is batched for efficiency anyway.

The speed of change replication is determined by the number of intermediate domains through which a change has to traverse. The fewer hops the update has to go through, the faster the replication. In the case of the tree topology, the traversal of the updates can cause significant latency for large trees where a recursive algorithm is used to locate and propagate updates. The speed of change replication is also determined by the frequency of data changes in each domain. WebSphere eXtreme Scale automatically adjusts the frequency of replication requests based on how often changes are found. The more changes, the more frequently replication occurs and the larger the batches that are replicated in a single interaction.

In summary, the replication performance is the best when there are minimum links between catalog service domains and a fairly steady rate of change; however, the fewer links can result in lowering the fault tolerance capability of the topology.

3.4.3 Collision management considerations

When separate updates occur at more than one location for the same record, an update collision will result when replication takes place. Replication occurs when a domain request changes from its foreign domains (also known as the “*pull*” design pattern of replication), so the collision is detected on the domain that is making the “*pull*” request. In most cases, WebSphere eXtreme Scale can internally resolve update collisions. It maintains an internal version number (actually a vector clock) for each grid entry. When an entry is replicated, if the version number coming in is higher than the one in the domain, the domain will be updated with the incoming entry. However, if both domains are updated close to simultaneously, or if the link between the domains is down when the updates occur, the same version number will be on both the incoming entry and the current entry. In this situation, a *collision arbiter* provides the logic to resolve update collisions between two domains.

WebSphere eXtreme Scale provides a default collision arbiter that determines how to handle collisions; however, this method is not the recommended practice. The default collision arbiter simply chooses the entry from the domain with the alphabetically higher domain name, which is not a viable solution for most production situations. Create your own custom arbiter (implementing the CollisionArbiter interface) to ensure that collisions will be handled in a sensible way.

The logic in a custom arbiter is often simple and often is similar to the logic in an OptimisticCallback that is used when you choose a lockStrategy of “OPTIMISTIC”. A common choice is to compare a timestamp field in the two entries (the timestamp might be in the key or the value object of the entry); as long as the clocks in each domain are sufficiently in synch, this logic is both simple and effective.

In a hub-and-spoke topology, the hub acts a clearinghouse for updates that ensures consistency across all linked domains. The spokes all have the same custom arbiter with a “hub rules” policy. In a tree topology, the parent nodes are clearinghouses for children nodes.

Depending on the likelihood of collisions and the number of domains, a hub might experience a heavy load due to arbitration. If this load is an issue, a tree topology is a better choice than a hub-and-spoke topology.

In topologies without a “clearinghouse”, collisions are resolved by each pair of domains. This approach can lead to the case where a collision happens and is resolved between two domains, then that result can lead to a collision with another domain, and so on.

3.4.4 Testing the topology

As with any design, it is a good idea to create a proof of concept (POC) project, build a realistic load testing environment, and then test and improve your multi-master replication topology across data centers through a cyclical and refining process.

It is usually difficult to produce the situation where the collision arbiter is called; all normal situations are handled automatically, based on the internal version number that every grid-stored object is given by WebSphere eXtreme Scale. If you change an object in one domain, the second domain knows that this object is newer than the object version it has because of that internal version number. To test your arbiter, you must force a situation where the same object is changed on both domain A and domain B simultaneously. Here is a proven way to call CollisionArbiter in testing:

1. Start with domains A and B in synch while allowing enough time for them to automatically synch up.
2. Use **xsadmin** to break the link between A and B.
3. Update a specific object in A.
4. Update the same object in B.
5. Reestablish the multi-master replication link between A and B.

A and B synch up quickly and your arbiter is called, because both versions of the object are at internal version number X+1.

3.5 Port assignments and firewall rules

When a WebSphere eXtreme Scale topology and its clients are spread across firewall boundaries, you will want to maintain control over the port numbers that are configured for communication between the components. With relation to WebSphere eXtreme Scale, we are concerned with ports assigned to catalog servers and container servers.

WebSphere eXtreme Scale uses Internet Inter-Object Request Broker (ORB) Protocol (IIOP) to communicate between JVMs. The catalog service JVMs are the only processes that require the explicit configuration of ports for the IIOP services and group services ports (unless you use Secure Sockets Layer (SSL) for security, in which case, an SSL port must be explicitly configured for clients, catalogs, and containers). Other processes dynamically allocate ports, but these port assignments can be specifically configured using **startOgServer** or Java command-line parameters, property files, or programmatically. This section describes which ports you need to be aware of and where they are configured. See Figure 3-6 on page 43.

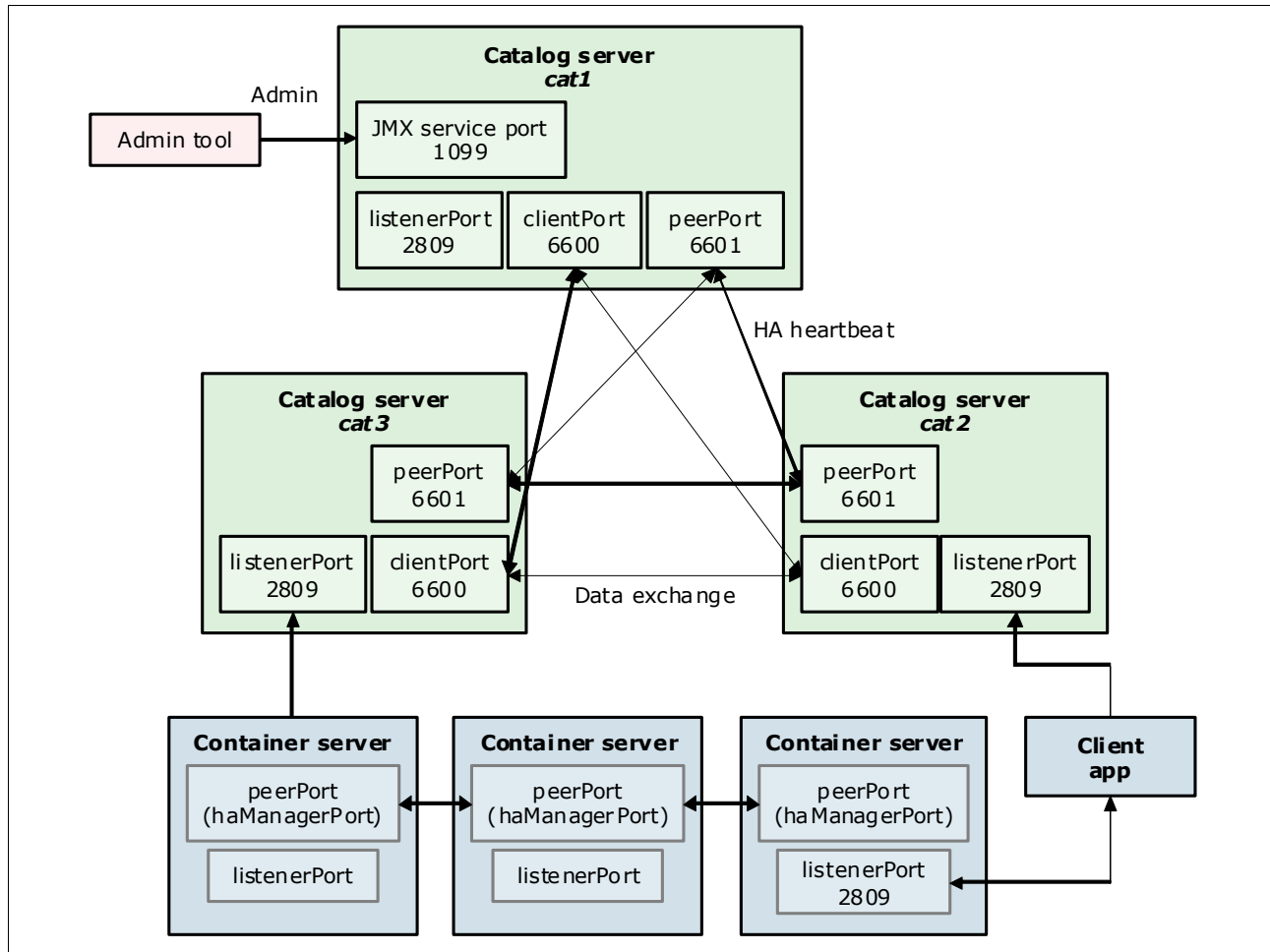


Figure 3-6 Catalog server domain ports

3.5.1 Catalog service domain ports

Catalog servers in a catalog service domain use ports for communication among themselves, with clients, and with the container servers. These ports are referred to as the *peerPort*, *clientPort*, and *listenerPort*. SSL security, if enabled, requires an additional port on every catalog server, container server, and client. In addition, the Java Management Extensions (JMX) service port is the interface for administration clients. As for any port, each JVM on a given host (using a given network interface card) must have a unique value for each of these ports.

peerPort, haManagerPort

The *peerPort* is used by the high availability (HA) manager to communicate between peer catalog servers in a cluster. When the catalog server is a managed server (running in WebSphere Application Server), this port is referred to as the *haManagerPort*. Think of this port as the “heartbeat” port.

HA manager and core groups: The HA manager and core groups are WebSphere Application Server features that are used by WebSphere eXtreme Scale for health checks and for intercommunication among catalog servers. These features are critical to the operation of the catalog server domains, and so, the jar file for the HA manager is included with stand-alone WebSphere eXtreme Scale installations.

For managed grids, you can use the normal WebSphere Application Server techniques to assign servers to core groups. For stand-alone grids, the WebSphere eXtreme Scale code automatically creates core groups in a completely transparent manner.

For stand-alone, WebSphere eXtreme Scale creates a separate core group for roughly every 20 servers. You can influence core groups in stand-alone grids by defining zones; servers in separate zones also are placed in separate core groups.

For stand-alone servers, the default for `peerPort` is 6601. To configure this port in stand-alone installations, use the **-catalogServiceEndpoints** command-line option when starting the catalog server:

```
startOgServer -catalogServiceEndpoints server:host:clientPort:peerPort,  
server:host:clientPort:peerPort, server:host:clientPort:peerPort
```

In WebSphere Application Server, the `peerPort` is the same as the `DCS_UNICAST_ADDRESS`. In the administrative console, you can see or change this port at **Servers → Server Types → WebSphere application servers → catalog_server → Ports**.

clientPort

The `clientPort` is used by catalog servers to access catalog service data on other catalog servers. The default is 6600.

To configure the `clientPort` in stand-alone installations, use the **-catalogServiceEndpoints** command-line option when starting the catalog server:

```
startOgServer -catalogServiceEndpoints server:host:clientPort:peerPort,  
server:host:clientPort:peerPort, server:host:clientPort:peerPort
```

In WebSphere Application Server, configure this port from the administrative console by selecting **System administration → WebSphere eXtreme Scale → Catalog service domains**.

listenerPort

The `listenerPort` defines the ORB listener port for containers and clients to communicate with the catalog service through the ORB. The default is 2809.

To configure the `listenerPort` in stand-alone installations, use the **-listenerPort** command-line option when starting the catalog server:

```
startOgServer -listenerPort port
```

In WebSphere Application Server, configure this port:

1. Take the default (inherited by the `BOOTSTRAP_ADDRESS` port configuration).
2. Use the administrative console to override the default:

System administration → WebSphere eXtreme Scale → Catalog service domains

Example: Starting a stand-alone catalog service domain

When starting a catalog server, you need to provide the host name and port of each of the catalog servers in the cluster.

When using the **startOgServer** command to start each server, use the **-catalogServiceEndpoints** command to specify the port information, for example, to start a three member catalog service domain consisting of servers cat1, cat2, and cat3:

```
startOgServer.sh cat1 -listenerPort 2809 -catalogServiceEndpoints  
cat1:hostA:6600:6601, cat2:hostB:6600:6601, cat3:hostC:6600:6601
```

```
startOgServer.sh cat2 -listenerPort 2809 -catalogServiceEndpoints  
cat1:hostA:6600:6601, cat2:hostB:6600:6601, cat3:hostC:6600:6601
```

```
startOgServer.sh cat3 -listenerPort 2809 -catalogServiceEndpoints  
cat1:hostA:6600:6601, cat2:hostB:6600:6601, cat3:hostC:6600:6601
```

For the **catalogServiceEndpoints** parameter, a comma-separated list is passed where each list element is specified as **<catalog server name>:<host name>:<client port>:<peer port>**. The ports are for inter-catalog server communications with 6600 and 6601 being the default values.

3.5.2 Container server ports

The container server ports to note are the HA manager port and ORB listener port. By default, these ports are dynamically assigned.

To use specific ports in a stand-alone JVM, you can configure the port numbers using options in the **startOgServer** command.

haManagerPort

The haManagerPort is used for internal communication between peer container servers running in WebSphere Application Servers. The haManagerPort value is inherited from the DCS_UNICAST_ADDRESS value for the application server.

listenerPort

The listener port is used for IIOP communication between peer container servers, catalog servers, and clients.

By default, this port is dynamically selected at start-up, but you can configure a specific port with the **-listenerHost** and **-listenerPort** command-line options when starting the container server.

In WebSphere Application Server, the listenerPort value is the BOOTSTRAP_ADDRESS value for each application server. In the administrative console, you can see or change this port at **Servers** → **Server Types** → **WebSphere application servers** → **catalog_server** → **Ports**.

Example for stand-alone container servers

In this example, the container server containerA is started on hostD. It assumes the three catalog servers on hostA, hostB, and hostC are listening on port 2809. It defines its own listenerPort as 2809.

```
startOgServer.sh containerA
  -catalogServiceEndpoints hostA:2809,hostB:2809,hostC:2809
  -haManagerPort port
  -listenerPort 2809
```

3.5.3 Client configuration

Clients will need to be aware of the ports that are required to connect to the catalog service domain. You might also need to configure a port for the client to listen on for callbacks from the container server.

Ports for connecting to the catalog service domain

WebSphere eXtreme Scale clients connect to the catalog service domain using the `ObjectGridManager.connect()` API. To perform the connection, the clients must know the listener endpoints for the catalog service domain. After they are connected, the clients retrieve endpoints for the container servers automatically from the catalog service.

To connect to the catalog service, the client passes the list of listener host:port pairs for the catalog servers to the connect API:

```
hostA:2809,hostB:2809
```

A list is important, because certain catalog servers might be down due to maintenance or a failure; the WebSphere eXtreme Scale client code will try each catalog server in the list until a successful connection is made. When connecting to a catalog service domain hosted within the same WebSphere Application Server cell, the client connection will automatically find the default catalog service domain by using the following API call on the `ObjectGridManager`:

```
connect(securityProps, overrideObjectGridXml)
```

If the default catalog service domain is external to the cell, or for stand-alone catalog servers, the catalog service endpoints must be specified using the following method on the `ObjectGridManager` API:

```
connect(catalogServerAddresses, securityProps, overrideObjectGridXml)
```

If the default catalog service domain is defined in the cell, the `CatalogServerProperties` API can be used to retrieve the catalog server addresses. The `XSDomainManagement` administrative task can also be used to retrieve any configured catalog service domain endpoints.

Ports for callback

A client can also receive callbacks from container servers when using the DataGrid API (for example, a `MapGridAgent`). To specify the port and network adapter to receive callbacks, set the `listenerHost` and `listenerPort` properties in the client properties file.

3.5.4 Multi-master replication ports

If you configure multi-master replication, you will need to configure ports for communication between the domains. These ports are part of the multi-master domain configuration in your `server.properties` file (via the `endpoints` property). If you do not configure SSL between

domains, the ORB listenerPort must be used. If you configure SSL between domains, the CSIV2 SSL Server Authorization Listener Address port must be used.

3.5.5 SSL ports

When security is enabled, a Secure Socket Layer (SSL) port is a required addition to the ports that were listed previously. The *SSLPort* is the secure port peer to the listener port.

For stand-alone clients, container servers, and catalog servers, use the JVM argument `-Dcom.ibm.CSI.SSLPort=<sslPort>` to define the secure listener port.

For clients, container servers, and catalog servers running in WebSphere Application Server, the transport security is managed by the WebSphere Application Server Common Secure Interoperability Protocol Version 2 (CSIV2) transport settings.

For more information about enabling and configuring security for WebSphere eXtreme Scale, see “Securing the Deployment Environment” in the *WebSphere eXtreme Scale Administration Guide*:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extrmescale.admin.doc/txsconfigsec.html>

3.5.6 Summary

Table 3-1 shows a summary of the ports that we have described.

Table 3-1 Port configuration summary

Connection	“caller” configuration	“destination” configuration
Client connects to catalog service domain's listenerPorts	Client passes the list of catalog service domain listener <i>host:port</i> values as a parameter of <code>ObjectGridManager.connect()</code> .	Catalog service domain (managed): The listenerPort is the BOOTSTRAP_ADDRESS port for the server (default) or can be specified in the admin console under Catalog service domains . Catalog service domain (stand-alone): The listenerPort is configured with <code>startOgServer -listenerPort port</code>
Container server connects to catalog service domain's listenerPorts	Container server (managed) Container server specifies the list of listener <i>host:port</i> values for the catalog service domain. Container server (stand-alone) Container server specifies the list of listener <i>host:port</i> values for the catalog service domain with <code>startOgServer -catalogServiceEndpoints host:port</code>	
Client (or another container) connects to the container server's listenerPort	The port numbers are retrieved by the client from the catalog service domain.	Container server (managed): The listenerPort is the BOOTSTRAP_ADDRESS port for the server (default). Container server (stand-alone): The listenerPort is configured with <code>startOgServer -listenerHost host -listenerPort port</code>

Connection	“caller” configuration	“destination” configuration
Catalog server to catalog server peer communication for high availability via the peerPort/haManagerPort	Catalog servers (managed): DCS_UNICAST_ADDRESS for the catalog server Catalog servers (stand-alone): start0gServer -catalogServiceEndpoints	
Catalog server to catalog server peer communication to access catalog data via the catalog server clientPort	Catalog server (managed): The clientPort is configured in the admin console under Catalog service domains . Catalog server (stand-alone): start0gServer -catalogServiceEndpoints	
Container server to container server communication for high availability via the container server haManagerPort. WebSphere Application Server environments only.	Catalog server (managed): DCS_UNICAST_ADDRESS for the container server	
Domain to domain links in a multi-master replication topology.	Configure the ports in the server.properties file via the endpoints (host:port) property: <ul style="list-style-type: none"> ▶ SSL: Port is the CSIV2 SSL Server Authorization Listener Address port. ▶ Non SSL: Port is the ORB listenerPort. 	



Capacity planning and tuning

When deploying WebSphere eXtreme Scale, we often need to perform sizing and capacity planning based on client requirements and usage scenarios.

This chapter includes the following topics:

- ▶ Planning for capacity
- ▶ Tuning the JVM

4.1 Planning for capacity

This section shows you how to plan for the hardware and Java virtual machine (JVM) characteristics that are required for your WebSphere eXtreme Scale topology. After you have completed the planning process, you can, if necessary, enhance your topology by organizing zones or by duplicating the topology at multiple data centers using multi-master replication or other topology-related tasks.

As for all capacity planning, the topology resulting from the planning must be validated by a proper performance test to verify that it achieves your performance targets.

4.1.1 Planning for catalog servers

We have described how to determine the number of catalog servers to cluster together (three or more in most cases). The memory and CPU requirements for catalog servers do not vary with the size of the data that is stored in the grid. A catalog uses memory as any Java code does, but it also uses an extremely small grid of its own to store routing tables and other internal data. The size of this data depends on the number of shards but not on the number of objects (keys and values) in those shards. Therefore, the memory that is used by a catalog server tends to be small and similar for all grids.

The recommended starting point for catalog servers is to use the Java virtual machine `-Xms` and `-Xmx` options equal to 768 MB. Any medium-capacity CPU will suffice for a catalog server. It is important that no catalog server compete with any container server (or any other application) for memory, CPU capacity, or bandwidth. During times of stress, it is critical that the master catalog server is able to quickly respond to container failures and other problems. If a container encounters severe memory or CPU starvation, the last thing that you want is for the catalog responsible for recovering the data to also be starved. For this reason, the leading practice is to place your three catalog servers on three separate modest-capacity hosts.

Container server planning: The remainder of this section addresses planning for container servers exclusively.

4.1.2 Planning for container servers: Building blocks

Grids, partitions, and shards are the major building blocks when performing capacity planning for WebSphere eXtreme Scale. Maps are important mainly as a way for you to determine the size of the data that you will store in the grid; you will need to sum the data that is stored in each map in your grid to arrive at a total size of data that you want to store:

- ▶ A *grid* divides the data set into partitions.
- ▶ Each *partition* holds an exclusive subset of the data. The data can be partitioned based on the key and the data for a partition is stored at run time in a set of shards.
- ▶ A *shard* is the smallest unit ObjectGrid can add or remove from a JVM. There is one shard for the primary copy of that data subset and possibly other shards for replicas.

The relationships among grid, partition, and shard is illustrated in Figure 4-1 on page 51. A grid has one or more partitions. Each partition has one primary shard, and optionally, additional replica shards. The term *grid* can mean one or more than one instance of an ObjectGrid object, its mapSets, and its associated BackingMaps.

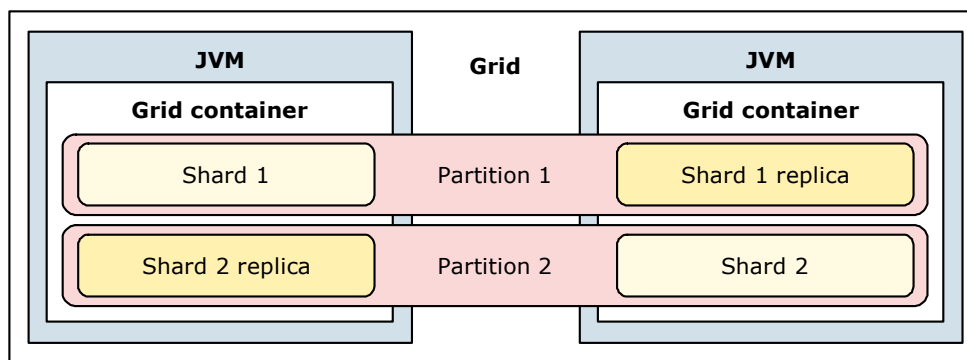


Figure 4-1 The relationships among grid, partition, and shard

4.1.3 What do we need to size when performing capacity planning

With a given data set size, there are a number of values that we want to determine as a result of the sizing exercise for the container servers:

- ▶ Memory requirements
- ▶ Number of JVMs to host the grid
- ▶ Heap size for each JVM
- ▶ Number of partitions
- ▶ Number of servers
- ▶ Minimum number of required CPUs

The values in this list are related to WebSphere eXtreme Scale. If this planning is a part of a bigger solution, additional factors need to be accounted for and the solution must be sized as a whole.

The following list contains a summary of the required steps to estimate the hardware, JVM, and WebSphere eXtreme Scale configuration when performing capacity planning:

1. Collect the following information about the data (Java objects) to be stored in WebSphere eXtreme Scale:
 - Maximum or peak number of objects
 - Average object size
 - The amount of redundancy that is required (number of replicas)
 - Estimated annual growth rate percentage for the next three to five years

The purpose of this step is to work out the primary data size, the total required object memory, and the total required physical memory.

2. Determine the memory requirements and partition count.

The purpose of this step is to size the memory requirement, the maximum heap size for each JVM, the number of partitions, and the number of servers.

WebSphere eXtreme Scale stores data within the address space of JVMs. Each JVM provides processor space for servicing create, retrieve, update, and delete calls for the data that is stored in the JVM. In addition, each JVM provides memory space for data entries and replicas. Java objects vary in size; therefore, it is always best to take a measurement to make an estimate of how much memory you need.

With good hash/partitioning logic for your keys, WebSphere eXtreme Scale will store roughly the same number of objects in each shard. Therefore, you ensure balanced memory use per JVM by ensuring a balanced number of shards per JVM. Choose a good value for the `numberOfPartitions` parameter in your deployment XML. The

`numberOfPartitions` value divided by the number of JVMs controls the number of shards per JVM.

Ideally, choose the `numberOfPartitions` value so that the difference in the number of shards stored on any of two JVMs in the grid is less than approximately 10%, both in normal use and when various failures occur. The process that we describe in this chapter helps you achieve this result.

3. Determine the CPU size and server count.

When performing capacity planning, especially to work out the number of required physical servers for the application, there are two primary factors to consider:

- The memory required to store the data on a single physical server
- The throughput required to process the requests on a single physical server

The processing power that is provided by the CPU has a great impact on the throughput, assuming that there is enough physical memory and network bandwidth. Normally, the faster the CPU is, the greater the throughput that can be achieved. Unlike memory use, CPU use is often impossible to estimate to any useful accuracy; it is usually necessary to prototype and measure, whether for WebSphere eXtreme Scale or any other software. In WebSphere eXtreme Scale, the CPU costs include these costs:

- Cost of servicing create, retrieve, update, and delete (CRUD) operations from clients. This cost tends to be fairly small.
- Cost of replicating from other JVMs. This cost also tends to be fairly small.
- Cost of invalidating (usually small).
- Cost of eviction policy. For default evictors, this cost is usually small.
- Garbage collection cost. This cost is tunable using standard techniques.
- Application logic cost (for custom evictors, agents, loaders, and so on). This cost is usually the biggest factor, but it might again be modest with proper custom code design.

The server count resulting from this CPU sizing step might differ from the number resulting from the previous step, “Determine the memory requirements and partition count.” You will need to take the larger of the two server counts as the final number.

For example, if you need 10 servers for memory requirements but that number provides only 50% of the required throughput because of CPU saturation, you will need twice as many servers (20).

In a similar way, if you determine that you need two servers based on the CPU sizing exercise, but you need six servers based on the memory requirements, you will need six servers instead of two.

Now, we take a closer look at the sizing calculation process for the previous steps.

4.1.4 Calculating the required memory for the data in the grid

The first step in determining the maximum size of the data to be stored in WebSphere eXtreme Scale (known as *totalObjectMemory*) is to estimate the maximum number of objects (*numberOfObjects*) and the average size of each object (*averageObjectSize*). It is critical that this estimate will be based on real data. For example, if you want to size for a Product object containing a “description”, do not estimate using descriptions that all read “this is a typical description” but instead use real descriptions (which are probably much longer).

The value for `numberOfObjects` usually must be estimated based on the number of users, number of requests, or another estimate that relates to the number of objects needed at

usage peaks. If you have an existing application that you want to enhance with WebSphere eXtreme Scale, it might be possible for you to directly measure this number.

There are two ways to calculate `averageObjectSize`: directly and indirectly. Use these steps for a direct calculation:

1. Either use your existing application or write a small Java application capable of creating (*new'ing*) 10,000 (or another number big enough to average out variation) of your objects. Include both keys and values. Start this application with **verbose:gc** enabled.
2. The application will *new* a few key and value objects to ensure all classes have been loaded.
3. Force a GC by calling `System.gc()`. This action causes **verbose:gc** log output showing the current memory use.
4. The application will then new 10,000 key-value pairs.
5. Force a GC again. Again, **verbose:gc** log output shows the current memory use after the key-value pairs were created in memory.
6. Using the **verbose:gc** log output, subtract the heap size before new'ing the 10,000 objects from the heap size after new'ing them. This number is the memory used by 10,000 of your key-value pairs.
7. Calculate the average object size:
$$\text{averageObjectSize} = \text{memory used by 10,000 key-value pairs} / 10,000$$
8. If you plan to use indexes (sometimes called *alternate indexes*) with the MapIndex plug-in, perform a similar exercise for each index. An index entry consists of a field from your value object (for example, the "accountName" field) and the key for that value object (which might be AccountNumber). Instead of creating 10,000 key-value pairs, create 10,000 of a typical instance of the field for the index (for example, "String accountName") and 10,000 keys. Perform this exercise for each index that you plan to have.
9. Add all the average sizes together to form a single `averageObjectSize` that you will use later.

Use these steps for an indirect calculation (if you do not have Java objects yet but only database records):

1. Get the size of each database record and determine the average size. If your Java objects will contain only certain columns from the database records, only count those columns. If your Java objects will contain columns from several tables, count data from all those tables. For fixed size records, data size is a consistent value.

For variable length records, analyze realistic data of varying sizes and average the data to get a true picture of `averageObjectSize`. For example, for VARCHAR columns, do not assume they are all 32 K in length; actual realistic data might average only a few hundred bytes and this difference is significant in your sizings when we later multiply that difference by 10 million objects.

2. WebSphere eXtreme Scale stores objects in key-value pairs. It is common that the key field in a database record is a tiny fraction of the size of the value object and can be ignored.

If the size of the key is over a couple of percent of the size of the value, modify the `averageObjectSize` to be the sum of the average key object size and the average value object size. You might count a given column twice, once as the key and once as a field in the value; if the key column is not needed as useful data in the value object, do not count it twice.

$$\text{averageObjectSize} = \text{averageKeyObjectSize} + \text{averageValueObjectSize}$$

After we have the value of the `numberOfObjects` and `averageObjectSize`, we can complete the memory size calculations:

1. Calculate the memory required for this map:

```
primaryObjectMemory = numberOfObjects * averageObjectSize
```

2. Repeat step 1 for each map in this grid.
3. Add all `primaryObjectMemory` numbers together to come up with a `primaryObjectMemory` for all maps in this grid, which will be used for all subsequent steps.
4. Calculate the maximum size of the data to be stored, which includes primaries and replicas (it does not matter whether they will be synchronous or asynchronous replicas):

```
totalObjectMemory = primaryObjectMemory + primaryObjectMemory *  
numberOfReplicas
```

4.1.5 Determining the maximum heap size and physical memory per JVM

Next, we want to know how much object memory will be taken up by our object data (primaries and replicas) in each of our container JVMs. WebSphere eXtreme Scale places objects evenly across our *C* containers (pick an arbitrary *C* to start with, because we can validate and fix it up later if necessary).

```
containerObjectMemory = totalObjectMemory / C (do not round anything yet)
```

Determining maximum heap size

Use `containerObjectMemory` to determine the *maxHeapSizePerJVM*. The calculations will take the following information into account:

- ▶ The `containerObjectMemory` value will be less than the maximum heap size of the JVM, because WebSphere eXtreme Scale itself (and WebSphere Application Server if your containers are in a managed installation topology) uses heap memory. We will account for this situation with the following variables:
 - *WXS-Footprint* = WebSphere eXtreme Scale runtime memory consumption in the JVM. As of WebSphere eXtreme Scale V7.0 or V7.1, *WXS-Footprint* = 100 MB (roughly).
 - *WAS-Footprint* = WebSphere Application Server runtime memory consumption in the JVM. As of WebSphere Application Server V6.1, *WAS-Footprint* = 300 MB (roughly).
- ▶ You are more likely to avoid Java `OutOfMemoryErrors` if you run at 60% of the maximum heap size for your JVM; you might be able to push this limit higher, depending on your particular situation.

We will use a 1.7 multiplier to represent the 60% heap utilization factor.

So, you can calculate the maximum heap size for the JVMs with the following formula:

```
maxHeapSizePerJVM = (containerObjectMemory + WXS-Footprint + WAS-Footprint ) * 1.7
```

Adjusting footprint estimates for the class data sharing feature

As of Java Platform 2, Standard Edition (J2SE) 5.0, for most Java Developer Kit providers, a Java feature called *class data sharing* might be significant here. This feature means that, within a single node, if the same classes are loaded in more than one JVM, that only one copy is held in memory and shared among all the JVMs. For us, that means that the *WAS-Footprint* (which is mostly classes of executable code) only has to be counted for one JVM in each node. Continue to count the *WXS-Footprint* for each JVM, because it is mostly container-specific data. Confirm that your JSE or Java Platform, Enterprise Edition (JEE)

version has this Java feature. WebSphere eXtreme Scale V6.1 and beyond on all but Solaris has this feature, because it uses the IBM Java Developer Kit. WebSphere Application Server V6.1 and beyond on all but Solaris also have it for the same reason. Recent Solaris Java Developer Kits have it, but you need to verify your specific Java Developer Kit. After you know your Java Developer Kit has this feature, adjust your math accordingly.

Adjusting max heap size for WebSphere eXtreme Scale agents and plug-ins

If you will be running WebSphere eXtreme Scale agents or loaders or other plug-in code that executes on the WebSphere eXtreme Scale container servers, you might need to increase the maximum heap size to allow for the memory that they will need when running.

Adjusting max heap size for 64-bit operating systems

With 64-bit OSs, you might not have to make an adjustment to the calculated memory numbers, such as `maxHeapSizePerJVM`. There is a 64-bit JVM feature, which is called *compressed references* (CR), that allows 64-bit object references (usually an extremely large part of memory use) to take up not double the space of a 32-bit reference but in fact the exact same space as a 32-bit reference. There is a slight CPU cost for this feature. See the following link for an excellent article about CR. The article is phrased in terms of WebSphere Application Server, but it applies to any use of the IBM JDK V6:

- “IBM WebSphere Application Server V7 64-bit performance: Introducing WebSphere Compressed Reference Technology”
ftp://public.dhe.ibm.com/software/webserver/appserv/was/WAS_V7_64-bit_performance.pdf

For 64-bit JVMs using IBM JDK V6 and later that have compressed references enabled, you can leave your total object memory calculations as is, provided that your JVM heap size (`-Xmx`) is less than 28 GB.

Using CR technology, IBM JDK using JVM instances can allocate heap sizes up to 28 GB with the same physical memory overhead as an equivalent 32-bit deployment. Beyond 28 GB per JVM, every reference will take twice the memory, but data values, such as extremely long string or byte array values, will not.

For non-IBM JDKs and the IBM WebSphere Application Server-supplied Sun JDK prior to WebSphere Application Server V7.0.0.9 (neither of which have CR), references also take twice the memory.

The recommendation for estimating memory use is that, if your JVM does not have CR, you double whatever memory sizes you calculate. Of course, if you used the direct method to measure memory use as described in 4.1.4, “Calculating the required memory for the data in the grid” on page 52 (creating 10,000 objects and using `verbose:gc`), you have measured using your actual object references, however many bits they are.

Rounding to the nearest 256 MB

For 32-bit or 64-bit JVMs, there is a slight performance advantage in `-Xmx` (the parameter that controls the maximum size of the Java heap) being a multiple of 256 MB. For this reason, round your maximum heap size value to the nearest 256 MB if possible. This value will be the value that you hope to use for the `-Xmx` option for your container JVMs. It might need to be adjusted depending on certain factors, primarily any limits on the effective maximum value for `-Xmx`.

Determining the OS tax

Physical memory consumption per JVM will differ depending on the operating system. There is a certain amount of memory needed by the operating system in addition to the Java heap size (referred to here as an *OS tax*). This tax includes the Java native heap space. The native heap is that portion of the OS process memory space needed by the JVM code itself, which is separate from the needs of the Java code running in the JVM. Java native methods (also known as Java Native Interface (JNI) methods), which are native code methods that are designed to be called from Java, also use memory from the native heap. The Java heap and the native heap both must come out of the total memory that is addressable for that OS process, which is limited for 32-bit OSs.

Table 4-1 lists the current OS tax for the common 32-bit operating systems. OS tax numbers with an asterisk (*) are estimates, because no recommendations from the JDK team are available for these numbers. The appropriate size for the Java native heap varies with the application (and especially if you have complex JNI methods), but that values that are shown in the OS tax column are a good starting point. If you find, during performance testing, that you need more, adjust accordingly; the typical symptom for running out of native heap is an `OutOfMemoryError` when your `verbose:gc` logging shows that you have plenty of Java heap space.

Table 4-1 shows the OS tax, the recommended maximum value to use for `Xmx` (obtained by subtracting the tax from the effective maximum memory size per process on that OS) and the Java heap memory available for data, assuming that you want to use no more than the recommended 60% of that maximum `Xmx` value at your expected peak load.

Table 4-1 Operating system tax for 32-bit operating systems

Platform	Type	OS tax for 32-bit JVMs	Effective max heap size	Available heap @ 60% use
AIX®	Automatic	0.75 GB	2.5 GB	1.5 GB
Linux		0.5 GB	1.5 GB	0.9 GB
Linux	hugemem	0.5 GB	2.5 GB	1.5 GB
Solaris		1.3 GB*	3.3 GB	1.8 GB
Windows		0.3 GB	1.5 GB	0.9 GB
Windows	/3 GB	0.3 GB	1.8 GB	1.1 GB
z/OS		1.3 GB	1.7 GB	1 GB

For 64-bit OSs (and JVMs), there essentially is no hard limit on OS process memory or max heap size. The theoretical maximum is 16 exabytes (1 million terabytes) and it will be a while before you can order that much physical memory on a single node. The only limit on heap size is determined by the garbage collection (GC) time that you can tolerate, which must be measured experimentally. Experience has shown, however, that a heap size of 5 - 6 GB usually can be tuned to a full GC time of around 2 - 300 ms on fairly modern CPUs.

With 64-bit OSs, you still need to apply the OS tax, but you will not ever have to reduce your Java heap to make room for native heap as you might in 32-bit. With a 64-bit process, you can give the Java heap and the native heap as much room as they each need (up to the total limit of 1M TB per process – effectively infinity today).

The OS tax for 64-bit operating systems differs in another way. Because essentially all of it is the Java native heap and because the JVM feature of compressed address references does not apply to the Java native heap, *the OS tax for 64-bit operating systems is double the figure in Table 4-1 on page 56.*

Adjusting the number of containers

Based on all these facts, we now validate and calculate the numbers we have so far. First, adjust the number of containers (*C*) if necessary:

- ▶ If your calculated `maxHeapSizePerJVM` is less than but close to the effective max `Xmx` value, you are fine (remember your `maxHeapSizePerJVM` already has 60% heap use built into it).
- ▶ If your calculated `maxHeapSizePerJVM` is well under the effective max `Xmx` value, consider reducing *C* and repeat the calculations, or you can leave *C* as is and simply allow a large amount of room for future growth. If you are well under the 5 - 6 GB size for 64 bit, you might prefer to stay well under to gain faster GC pause times.
- ▶ If your calculated `maxHeapSizePerJVM` is greater than the effective max `Xmx` value, increase *C* and repeat the previous calculations.

Determining the physical memory per JVM

Take your `maxHeapSizePerJVM` and add the appropriate OS tax, which gives you the physical memory that is required for each container JVM.

`physicalMemoryPerJVM = maxHeapSizePerJVM + OS tax`

For more information about sizing the Java heap, please refer to “Sizing the Java heap” at this website:

http://publib.boulder.ibm.com/infocenter/javasdk/tools/index.jsp?topic=/com.ibm.java.doc.igaa/_1vg00014884d287-11c3fb28dae-7ff6_1001.html

Determining the number of servers (also known as hosts or boxes)

The preferred practice from a general operations point of view is to have an equal number of container JVMs on each server; this approach simplifies server-level configuration and balancing. The main criteria for how many JVMs a server can host is how much physical memory the server has (CPU capacity is another factor). The dollar price per server is usually tied to its physical memory and the number of processors and cores; often, the amount of memory is proportional to the number of processors as well.

Therefore, you can determine the number of servers that you need, which we will call *N*, by dividing your total physical memory that is needed by the memory in one server (total physical memory needed is `physicalMemoryPerJVM * C`). Of course, you can use more servers if you want, because you do not have to use up all the physical memory on each server. You can also share servers that host other applications if other factors (relative stability of the applications, server maintenance schedules, organizational issues, and so on) permit.

The absolute minimum for a high availability environment is two servers (*N*=2) so there is no single point of failure (SPOF). Preferred practice is to have *N*=3 or 4, so that the loss of a server represents the loss of only 25% of your total memory and CPU (1 of 4) and not half (1 of 2). For an extremely small grid, you might go with *N*=2 servers at this point in the calculations; in the next section on failure handling, you will probably decide to add one more.

Important: Do not use virtual memory (or memory paging) on a server hosting JVMs. This rule is a general Java principle, not something specific to WebSphere eXtreme Scale. We used the term “physical memory” in our discussions for this reason.

4.1.6 Topology considerations for failure handling

Now, you must account for failures. This part is always a judgment call, but there are three failure scenarios that are commonly considered:

- ▶ The topology must survive a single JVM failure and be able to reconstruct the full set of primary and replica shards (`maxSynchReplicas` or `maxAsynchReplicas`). The simplest and best way to allow for this reconstruction is to increase *C* by 1. For this reason, we did not previously round up *C* so it was evenly divisible by *N*, your number of servers. After increasing by 1, if the new *C* is not evenly divisible by *N*, you might further increase *C* until it is. Extra memory never hurt any grid.
- ▶ The topology must survive a single server failure and be able to reconstruct the full set of primary and replica shards (`maxSynchReplicas` or `maxAsynchReplicas`). This rule allows any one server to be down for an extended period of time, such as for maintenance or diagnostics. To ensure this capability, add one more server with the same amount of memory as the others (if servers vary in memory size, add one more of the larger servers).

If you want to be less conservative, you can decide that your requirement is only that the grid remain operational. If your topology to this point has allowed for one or more replicas, one server going down will result in the (extremely temporary we hope) loss of a number of replicas (and a warning message), but application operation will continue because only primaries are used in operation. Get that server running again immediately so that a subsequent JVM failure will not trigger the loss of actual primary data.

- ▶ The topology must survive a single server failure followed by a single JVM failure. If you have shut down a server for the maintenance of the OS or another reason, a single JVM failing must not disable your grid. You can determine if you can survive this situation with the following conditions:
 - Normal state is *C* containers over *N* servers, or *C/N* containers per server.
 - One server down means you have *C* - *C/N* containers remaining.
 - A subsequent JVM down means that you have *C* - *C/N* - 1 containers remaining.
 - If and only if the following condition is true, will you survive a double failure:
$$(C - C/N - 1) * \text{Available heap @ 60\% use} > \text{PrimaryDataSize}$$
- And now, you know why we included the “Available heap @ 60% use” column in Table 4-1 on page 54.

4.1.7 Determining the number of partitions

The value that you choose for `numberOfPartitions` in the deployment XML is only significant if you choose a number that is too small or far too large. There are various subtle effects from choosing a large or a small number but the primary consideration results from the fact that a given partition must live wholly in a single container (just as a given object – or entity tree if using the WebSphere eXtreme Scale entity model - must live wholly in a single partition). eXtreme Scale is extremely elastic in that you can grow or shrink the number of containers (*C*) while the existing container servers (and the catalog servers) are running and in use. Each new container simply uses the same XML file pair as the previous containers did. The only limit on the number of containers you have running is the number of partitions; if `numberOfPartitions` is *P*, you cannot have more than *P* containers started to host data. Additional containers will start, but they will not receive any partition to host.

For certain applications, *C* (the number of containers calculated previously) is the number that you expect to always have started. For other grids, you know that *C* is fine most of the time but you periodically have spikes of load (quarterly, end of month, and so on) where you will

want to start more containers and leave them up for awhile. When calculating `numberOfPartitions`, consider the largest number of containers you will have running during anticipated peak loads (do not worry, we will build in a buffer for unanticipated peaks). We call this number of containers *maxC*; if your *maxC* equals your *C*, that is fine.

After you have *maxC*, you need to consider how many partitions you want per container. The main motivation here is that your number of partitions is not always (might never be) evenly divisible by your number of containers. When it is not evenly divisible, certain containers will have one less partition than other containers, which is due to the key fact that partitions are never split across containers. With 10 containers and 11 partitions (an unrealistic but informative example), one container will have two partitions and the others one partition. Data is spread evenly across the partitions (when WebSphere eXtreme Scale is tuned properly), which means that one container will have 50% more data than the others and thus need 50% more Java memory for its heap. Also, this one container will on average handle 50% more of the request traffic than the others. You can easily even things out by increasing the number of partitions.

A good rule is to have at least 10 partitions per container, because each container will typically be within 5% of the others in data/memory and traffic. Therefore, our choice for the number of partitions *P* is now 10 times *maxC*.

Note that WebSphere eXtreme Scale documentation mentions five partitions per container (5 partitions + the typical five replicas means 10 shards per container). Whether to choose five or 10 partitions (10 or 20 shards) per container is within the realm of honest debate and judgement. Twenty shards will mean that, in the event of a failure, there will be a more even spread of the partitions from the failed JVM or node to the remaining ones and thus also more even CPU use. Whether the added evenness is worth it to you is where the judgement comes in.

Consider 10 JVMs and 51 partitions (102 shards assuming one replica) versus 10 JVMs and 101 partitions (202 shards). For the former case, you have this situation:

- ▶ Normally, you have eight JVMs with 10 shards and two JVMs with 11 shards, which is a 10% imbalance in memory use.
- ▶ If a JVM goes down, you end up with seven JVMs with 11 shards and two JVMs with 12 shards, which is a 9% imbalance in memory use.

You have this situation in the latter case of 101 partitions (202 shards):

- ▶ Normally, you have eight JVMs with 20 shards and two JVMs with 21 shards, which is a 5% imbalance in memory use.
- ▶ If a JVM goes down, you end up with five JVMs with 22 shards and four JVMs with 23 shards, which is a 4.5% imbalance in memory use.

Another consideration is that each partition results in certain threads executing, such as for eviction or agent execution. Too high a value for `numberOfPartitions` can result in more threads running than your servers can support. This condition can only be judged by testing, but we can say that a small number of container JVMs (for example, eight) multiplied by 10 will still be a small number of such threads, while a large number of containers (for example, 128) multiplied by 10 is significantly more threads than if multiplied only by 5.

Whether you go with a multiplying factor of 5 or 10 to get your initial value, you want to round this value up to the next prime number. Certain operations in WebSphere eXtreme Scale are much more efficient if `numberOfPartitions` is a prime number. The advantage of using a prime number outweighs the disadvantage of uneven data spread as long as the variance is small (our 5% is considered extremely small).

So, the final equation for determining a good `numberOfPartitions` is:

`numberOfPartitions = (maxC * 5 or 10) and round up to the next prime`

Remember that you can always change this `numberOfPartitions` value in the future as long as you can restart all containers with the changed deployment XML.

4.1.8 Determining `numInitialContainers`

Compared to `numberOfPartitions`, determining `numInitialContainers` is easy. In that section, we talked about the difference between “*C*” and “*maxC*”. *C* is the number of container JVMs that you normally start. In nearly all cases, the preferred practice is to set `numInitialContainers` to *C*. As you start container JVMs, WebSphere eXtreme Scale will wait until the number started equals `numInitialContainers` before it begins placing (empty) partitions on those containers. If, after it begins placing partitions, you start more containers, WebSphere eXtreme Scale simply has to move several partitions to the new containers. This exercise is just a waste of time, CPU, and bandwidth and slows down your overall grid start-up. If you set `numInitialContainers` to *C*, the placement will happen once and correctly the first time.

Note that, prior to reaching `numInitialContainers` and placing partitions, the grid is not functional, which is correct. Clients trying to access the grid will get an exception. If you run the `xsadmin -mapsizes` command, you will see only a brief cryptic message instead of the list of containers and partitions that you expect. In this way, WebSphere eXtreme Scale ensures that the grid cannot be accessed until it is ready to work properly.

4.1.9 Determining the number of CPUs

Although memory is usually the major criteria for the number of servers that you need, you must not ignore processor (CPU) capacity.

Processor costs include the following items:

- ▶ Cost of servicing create, retrieve, update, and delete operations from clients
- ▶ Cost of replication from other JVMs
- ▶ Cost of invalidation
- ▶ Cost of eviction policy
- ▶ Cost of garbage collection
- ▶ Cost of application logic

To start, perform the following CPU sizing exercise for single partition transactions:

1. Use two servers and start the planned number of container JVMs on each server.
2. Use the calculated partition counts (`numberOfPartitions` in 4.1.9, “Determining the number of CPUs” on page 60).
3. Then, preload the JVMs with enough data to fit on these two servers only.
4. Using a separate server, or servers, as a client, run a realistic transaction simulation against this grid of two servers.

To calculate the baseline, try to saturate the processor usage. If you cannot saturate the processor, it is likely that the network is saturated. If the network is saturated, add more network cards (if possible) and round robin the JVMs over the multiple network cards.

If neither are saturated and yet more load does not increase CPU or network usage, start more clients. Remember, there is only one Object Request Broker (ORB) connection

between any one client JVM and any one container JVM regardless of how many threads the client is running; that connection has only so much capacity.

5. Run the computers at 60% processor usage, and measure the create, retrieve, update, and delete transaction rate.

This measurement provides the throughput on two servers. This number doubles with four servers, doubles again at eight servers, and so on. This scaling assumes that the network capacity and client capacity are also able to scale. As a result, WebSphere eXtreme Scale response time must be stable as the number of servers is scaled up. The transaction throughput must scale linearly as computers are added to the data grid.

Now, let us take a look at parallel transactions. Parallel transactions touch a subset of the servers. The subset can be all of the servers.

If the transaction touches all servers, the throughput is limited to the throughput of either the client initiating the transaction *or* the slowest server being touched. Larger grids will spread the data out more and provide more CPU, memory, network, and so on. But, the client must wait for the slowest server to respond *and* the client must then consume the results.

When the transaction touches M of N servers, the throughput is N/M times faster than the throughput of the slowest server, for example, with 20 servers and a transaction that touches five servers. The throughput is 4x (20/5) the throughput of the slowest server in the grid.

When a parallel transaction completes, the results are sent to the client thread that started the transaction. This client must then aggregate the results (if any) single threaded. This aggregation time increases as the number of servers touched for the transaction grows. However, this time depends on the application, because it is possible that each server returns a smaller result as the data grid grows.

Typically, parallel transactions touch all of the servers in the data grid, because partitions are uniformly distributed over the grid. In this case, throughput is limited to the first case.

4.1.10 An example of a WebSphere eXtreme Scale sizing exercise

Assume that we need to store four types of objects. Each of the four objects has the following average key sizes and value sizes:

- ▶ Customer: Avg. key = 10 bytes, avg value = 2 KB, peak number = 1,000,000
- ▶ Product: Avg. key = 4 bytes, avg value = 40 KB, peak number = 100,000
- ▶ Order: Avg. key = 8 bytes, avg value = 20 KB, peak number = 500,000
- ▶ OrderItem: Avg. key = 8 bytes, avg value = 8 bytes, peak number = 5,000,000

Our primaryObjectMemory value is then:

```
(2010 bytes * 1,000,000) +  
(40,004 bytes * 100,000) +  
(20,008 bytes * 500,000) +  
(16 bytes * 5,000,000) =  
16.1GB
```

Follow these steps for the sizing exercise:

1. We want to have a single replica for high-availability purposes so we will need to store 16.1 GB objects * 2, or totalObjectMemory = 32.2 GB in the grid including replication.

2. We want to place the objects evenly across eight containers ($C = 8$):

$$\text{containerObjectMemory} = 32.2 \text{ GB} / 8 = 4.025 \text{ GB}$$
3. Our grid will not run on WebSphere Application Server so we only need to account for the WebSphere eXtreme Scale footprint of 100 MB:

$$\text{maxHeapSizePerJVM} = (4.025 \text{ GB} + 0.1 \text{ GB}) * 1.7 = 7.0125 \text{ GB}$$
4. This value is larger than the generally recommended 5 - 6 GB for 64-bit Linux (our OS). We try again with $C = 11$ containers:

$$\text{containerObjectMemory} = 32.2 \text{ GB} / 11 = 2.927 \text{ GB}$$

$$\text{maxHeapSizePerJVM} = (2.927 \text{ GB} + .1 \text{ GB}) * 1.7 = 5.146 \text{ GB}$$
5. This value is within the recommended value. Round up to the next highest multiple of 256 MB, which is 5.376 GB ($256 \text{ MB} * 21$).
6. The OS tax for the 64-bit Linux platform is 0.5 GB. We have several native methods so we will use an OS tax of 1 GB.
 That means:

$$\text{physicalMemoryPerJVM} = 5.376 \text{ GB} + 1 \text{ GB} = 6.376 \text{ GB}$$
 or 6.5 GB rounded to make our math easier.
7. Now, we look of the number of servers and JVMs per server that are required to hold our data. We will round our C value (now 11) up to 12 so that it spreads evenly over our four available servers:
 - a. $\text{maxNumberJVMsPerServer} = 35 \text{ GB available memory per server} / 6.5 \text{ GB}$
 - b. $\text{physicalMemoryPerJVM} = 5 \text{ JVMs per server, with some remaining memory}$
 - c. $\text{numberOfServers} = 12 \text{ JVMs} / 4 \text{ JVMs per server} = 3 \text{ servers}$
8. Now, we account for the three failure scenarios:
 - a. One JVM fails:

$$12 \text{ container JVMs} - 1 = 11 \text{ JVMs.}$$

$$11 \text{ JVMs} * 2.927 \text{ GB ("containerObjectMemory")} = 32.2 \text{ GB}$$

32.2 GB is equal to or greater than our required totalObjectMemory. Note that this number worked out so nicely because we rounded C up from 11 to 12 earlier.
 - b. One server fails or is stopped for maintenance:

To ensure that the grid is unaffected, increase the number of servers from three to four and keep the number of JVMs per server the same value of 4. We will now have not 12 but 16 container JVMs with the previously calculated maxHeapSizePerJVM and physicalMemoryPerJVM value. Because they are spread across four servers, the amount of physical memory per server is unchanged.
 - c. One server + one JVM fail:

In this scenario, the number of container JVMs goes from 16 to 12 to 11. We know from the first failure scenario that 11 JVMs will still hold our 32.2 GB of primary and replica data and 40% unused heap space for emergencies. Therefore, we know our topology can handle this failure scenario as well.
9. Next, we will calculate numberOfPartitions. Our number of container JVMs, C , is 16 so we choose to use the "times 5" factor:

$$\text{numberOfPartitions} = 16 * 5 \text{ rounded up to prime} = 83$$

We say that each of our four servers is a dual-core CPU (total of eight cores).

Assume four servers can provide enough of the throughput that is required (that is, the number of servers from “4.1.9, “Determining the number of CPUs” on page 60” is *not* greater than 4), then we can fill out all the data in Table 4-2.

Table 4-2 Sizing requirements for data center 1

Value	Data center 1
Number of servers (physical machines)	4
Number of CPUs	4 * 2 = 8 cores
Total physical memory	26 GB * 4 servers = 104 GB
Number of WebSphere eXtreme Scale container JVMs	16
MaxHeapSize per JVM, also known as Xmx	5376 m or 5.376 GB
numberOfPartitions	83

4.2 Tuning the JVM

In general, few or no special JVM settings are required for a WebSphere eXtreme Scale environment. This section will provide additional guidance about JVM tuning options that can help improve performance.

4.2.1 Selecting a JVM for performance

WebSphere eXtreme Scale supports Java SE Version 1.4.2, and later, and works on 32-bit or 64-bit JVMs. The recommendation is to use the latest available version of Java Platform, Standard Edition for the best performance.

WebSphere Application Server V7 uses the IBM virtual machine for Java (IBM JVM) based on Java Platform, Standard Edition 6 (Java SE 6.0) on platforms other than Solaris and HP-UX. For Solaris and HP-UX, the Sun HotSpot JVM is used.

To streamline support issues, it is also recommended that you use the JVM from the WebSphere run time on any platform that WebSphere Application Server supports. Using a JVM from another vendor will require that you go to that vendor for JVM-related issues, plus you might miss out on key features, such as compressed references.

4.2.2 Tuning for efficient garbage collection

Garbage collection is the process of freeing unused objects so that portions of the JVM heap can be reused. Garbage collection is triggered automatically when there is a request for memory, for example when creating an object, and when the request cannot be readily satisfied from the free memory that is available in the heap (allocation failure). WebSphere WebSphere eXtreme Scale creates temporary objects that are associated with each transaction. Because these objects affect garbage collection efficiency, tuning garbage collection is important.

IBM JVM and Sun HotSpot garbage collection schemes

Although the garbage collection function that is provided by the Sun HotSpot and IBM garbage collectors is the same, the underlying technology differs.

For both JVMs, garbage collection takes place in three phases:

- ▶ Mark
- ▶ Sweep
- ▶ (Optional) Compact

The implementation of the garbage collection phases differs, mainly because the HotSpot engine is known as a *generational collector* and the IBM JVM (by default) is not.

In its default configuration, the IBM JVM consumes its entire heap before a garbage collection is triggered. With the HotSpot JVM, a garbage collection is triggered when either the nursery or the full heap is consumed. Whether a full heap garbage collection or nursery garbage collection is performed, the first phase is to mark all referenced objects in the region being collected. This method leaves non-referenced objects unmarked and the space that they occupy free to be collected and reused.

Following the mark phase, free chunks of memory are added to a free list. This phase is referred to as *sweeping*.

Occasionally, following the sweep phase a compact phase is performed. The compaction moves objects closer together to create larger contiguous free chunks. There are a number of triggers that can cause a compaction. For example, if after sweep, there is still not a large enough contiguous chunk of memory, compaction executes. Also, for most `System.gc()` calls, a compaction is done. Relative to the other phases involved, compaction can be a time-consuming process and needs to be avoided if possible. The IBM JVM is optimized to avoid compactations, because this process is an expensive process.

Setting the garbage collection policy

With the IBM JVM, you can select the garbage collection policy using the `-Xgcpolicy` JVM argument:

```
-Xgcpolicy optthruput | gencon | optavgpause | subpool
```

Use the following considerations to set the garbage collection policy:

- ▶ As a starting point, test with `optthruput`. One advantage of `optthruput` over `gencon` with respect to WebSphere eXtreme Scale is that you cannot use memory-based eviction with `gencon`. Memory-based eviction depends on WebSphere eXtreme Scale knowing when heap use reaches a configurable limit (default 70%); `gencon` uses several separate heaps that dynamically vary and thus monitoring heap use becomes impractically expensive.
- ▶ For high update rate scenarios (100% of transactions modify entries) and the IBM JVM, you might find that the `optavgpause` policy for garbage collection is best. With `optavgpause`, the garbage collector pauses become uniform and long pauses are not apparent.

The downside of this policy is that throughput is reduced by roughly 5% more than with other policies; for example, if `optthruput` results in 5% of every hour in GC pauses, `optavgpause` might result in 10% spent in GC pauses depending on the particular application. You only want to use `optavgpause` if `optthruput` does not give you acceptable GC pause times even after the tuning of other parameters, such as increasing `Xmx` or `Xms`.

- ▶ For scenarios where the data is updated relatively infrequently (10% of the time or less), use the `gencon` (generational) garbage collector. The generational schema attempts to achieve high throughput along with reduced garbage collection pause times.

Experiment with all collectors to determine the best collector for your environment. Run with verbose garbage collection (`verbosegc`) turned on to check the percentage of the time that is spent collecting garbage.

Consider these other factors:

- ▶ A read-mostly scenario performs even with large heaps (20 GB or more) if `optthruput` or `gencon` is used. However, after the tenure heap fills, a pause proportional to the live heap size and the number of processors on the computer occurs. This pause can be large on smaller computers with large heaps.
- ▶ Use more cores to reduce pauses in garbage collection. A physical server with eight cores has a faster garbage collection than a physical server with four cores. With 16 or more processors, the IBM JVM has a policy called `subpool`, which might yield better GC behavior.
- ▶ If you are using a Sun JVM, adjustments to the default garbage collection and tuning policy might be necessary.
- ▶ You can nearly always change your topology to have more container JVMs, each with a smaller heapsize; a smaller heapsize generally reduces GC pause times.

Monitoring garbage collection

You can determine how the heap is used by collecting a **verbosegc** trace. A *verbosegc trace* prints garbage collection actions and statistics to `stderr`. The **verbosegc** trace is activated using the Java runtime option of **verbosegc**. Output from **verbosegc** differs for the Java HotSpot and IBM JVMs, as shown by the following examples.

Example 4-1 provides a sample JVM **verbosegc** trace output.

Example 4-1 JVM verbosegc trace output

```
<AF[8]: Allocation Failure. Need 15727 bytes 5875 ms since last AF>
<AF[8]: managing allocation failure, action=1 (23393256)/131070968)
(2096880/3145728)>
<GC: Tue Dec 18 17:32:26 2011
<GC(12): freed 75350432 bytes in 168 ms, 75% free (100840568)/134216696>
<GC(12): mark: 129 ms, sweep: 39 ms, compact: 0 ms>
<GC(12): refs: soft 0 (age >= 32), weak 0, final 0 , phantom 0>
<AF[8]: completed in 203 ms>
```

Example 4-2 provides a sample Solaris HotSpot JVM **verbosegc** trace output (young and old).

Example 4-2 Solaris HotSpot JVM verbosegc trace output

```
[GC 325816K->83372K (776768K), 0.2454258 secs]
[Full GC 267628K->83769K <- live data (776768K), 1.8479984 secs]
```

Using the IBM JVM output that is shown in Example 4-1, we can understand the following information:

- ▶ In the first line, the metric following the word `Need` is the size of the failed allocation that caused the garbage collection. The same line includes the amount of time in milliseconds since the last allocation failure.
- ▶ The second line, with the `<AF[8]>` tag, displays the amount of free space in the heap and in an area of the heap that is referred to as the *wilderness*. The line reports 23393256 free bytes out of a possibly 131070968 bytes.
- ▶ The third line, `(2096880/3145728)`, refers to free wilderness area, which is usually ignored.

The next set of lines provides information about the garbage collection that was caused to satisfy the allocation failure. The first line is a time stamp. This line is followed by a line that includes the time to complete the garbage collection, 168 ms, and the amount of free space

after the garbage collection, 75%. Both of these metrics are extremely useful in understanding the efficiency of the garbage collection and the heap usage.

Following this line is a line describing the time for the various components of the garbage collection. Verify that the number following “compact” is normally 0. That is, a well-tuned heap avoids compactions. Finally, for the garbage collection, there is a line regarding soft, weak, and phantom references, as well as a count of finalizers. This information is then bracketed by a line with a time for the full allocation failure.

4.2.3 Increasing the ORB thread pool

WebSphere eXtreme Scale requires a working ORB to operate. You can use WebSphere eXtreme Scale with ORBs from other vendors. However, if you have a problem with a vendor ORB, you must contact the ORB vendor for support. The IBM ORB implementation is compatible with third-party JVMs and can be substituted if needed. The WebSphere eXtreme Scale product documentation describes how to use its “endorsed” directory contents to perform this substitution.

Make sure that you use the recommended ORB configuration in the WebSphere Application Server ORB implementation to ensure optimum performance in a managed grid. In particular, make sure that the thread pool for incoming requests is sufficient.

See the following resources:

- ▶ Using the Object Request Broker with stand-alone WebSphere eXtreme Scale processes
<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.e xtremescale.admin.doc/txsinstallorb.html>
- ▶ ORB properties
<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.e xtremescale.admin.doc/rxsorbproperties.html>
- ▶ orb.properties tuning
<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.e xtremescale.prog.doc/cxsjvmtune.html>

4.2.4 Thread count

The thread count depends on several factors. A shard is an instance of a partition, and can be a primary or a replica. With more shards for each JVM, you have more threads with each additional shard providing more concurrent paths to the data. Each shard is as concurrent as possible although there is a limit to the concurrency.

Each shard can potentially have several threads for separate purposes. There is a thread for normal grid access. There is a thread for executing an agent. There is a thread for evictors and for a loader. There is no explicit configuration parameter for the number of threads per container JVM; depending on which features you use, WebSphere eXtreme Scale will spawn one or more threads for that feature. It is your responsibility to ensure, through testing, that your servers have enough CPU capacity to support the features that you want to use and their threads.

You configure the number of threads in the container server threadpool by the `minThreads` and `maxThreads` properties in the `server.properties` file. WebSphere eXtreme Scale threads running grid access, agents, evictors, and many other tasks have thread names that begin with “WXS” so they can be easily identified in thread dumps.

4.2.5 Sources and references

The following articles in the WebSphere Application Server and WebSphere eXtreme Scale information centers can provide additional information:

- JVM tuning for WebSphere eXtreme Scale

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.e xtremescale.admin.doc/cxsjvmtune.html>

- Tuning the IBM virtual machine for Java (WebSphere Application Server V7 Network Deployment on a distributed platform other than Solaris and HP-UX)

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.nd.doc/info/ae/ae/tprf_tunejvm_v61.html

- Tuning HotSpot JVMs (WebSphere Application Server V7 Network Deployment on Solaris and HP-UX)

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.nd.doc/info/ae/ae/tprf_hotspot_jvm.html

The following paper also has excellent tips about setting heap sizes in WebSphere Application Server. It was written for WebSphere Business Integration products that use WebSphere Application Server as a base product, but the techniques and insight into JVM heap size calculations can be helpful:

- *WebSphere Business Integration V6.0.2 Performance Tuning*

<http://www.redbooks.ibm.com/redpapers/pdfs/redp4304.pdf>



Grid configuration

This chapter describes how configure a WebSphere eXtreme Scale environment.

The most common and recommended way to configure WebSphere eXtreme Scale is to use configuration files. Configuration files are relatively easy to write and they do not need any tool other than a simple text editor. This chapter will show you what these files are and how they are configured. This chapter also describes plug-ins that can be installed to optimize the WebSphere eXtreme Scale environment performance.

This chapter includes the following topics:

- ▶ Configuration overview
- ▶ Catalog service domain
- ▶ ObjectGrid plug-ins
- ▶ BackingMap plug-ins

5.1 Configuration overview

Configuring WebSphere eXtreme Scale is a subject that starts out relatively simple and can become much more complex as you use more features of the product. You can configure a simple functional grid using only two XML configuration files with under a dozen lines between them (only about five of which are unique to your grid). As you enhance your grid to use more features, the complexity grows -- as is true for any software product. The number of available options and plug-ins is significant, and a number of various configuration files can be used. This section provides an overview of those configuration files and how to use them. It includes the following topics:

- ▶ Server XML configuration
- ▶ Client XML configuration
- ▶ Server properties
- ▶ Client properties
- ▶ Externalizing the server XML configuration in WebSphere Application Server
- ▶ Duplicate server names in WebSphere Application Server

Figure 5-1 shows the configuration files that are available in a WebSphere eXtreme Scale environment (not including `security.xml`). The dotted line boxes are optional configuration files. Although the diagram shows an environment that is deployed on WebSphere Application Server, the configuration files are similar in a stand-alone environment. The major difference is that in a stand-alone environment, you can configure the Object Request Broker (ORB) through the `orb.properties` configuration file. When running in WebSphere Application Server, you must configure the ORB using the administrative tools.

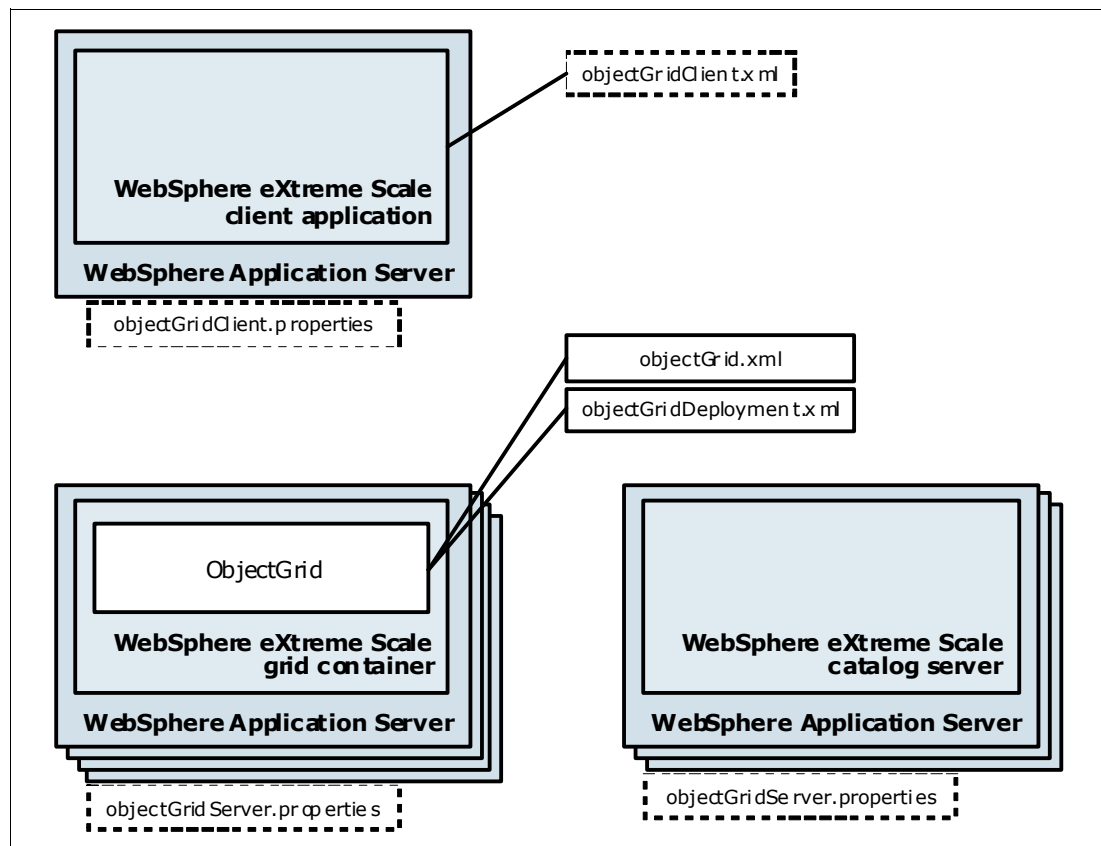


Figure 5-1 Overview of WebSphere eXtreme Scale configuration files

If you are a novice with using WebSphere eXtreme Scale configuration files, use a specialized editor, such as the editor that is provided with the Eclipse platform to begin. Using such an editor will verify that the XML is well formed. It is frustrating to deploy an application start or restart a cluster and see that there are XML errors in the WebSphere eXtreme Scale configuration files.

5.1.1 Server XML configuration

The WebSphere eXtreme Scale grid containers host the object grids. The following XML configuration files are required:

- The ObjectGrid descriptor XML file

You use this file to define and configure one or more object grids. Typically, the file is referred to as the `objectGrid.xml` configuration file. For details and examples, see this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.websphere.extremescale.admin.doc/txsreblmaps.html>

Validating the file: You can validate the ObjectGrid descriptor XML configuration file using the schema that is available at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extremescale.admin.doc/rxslclschema.html>

- The ObjectGrid deployment policy XML file

This file is required for all but the simplest local grids. You use this file to define how the object grid or grids that are defined in the ObjectGrid descriptor file are deployed in the grid containers at run time. For example, you can define the number of replicas and implement zoning rules in this file. Typically, this configuration file is referred to as the `objectGridDeployment.xml` configuration file. For details and examples, see this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.websphere.extremescale.admin.doc/rxsdpolicyref.html>

Validating the file: You can validate the ObjectGrid deployment policy XML configuration file using the schema that is available at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.websphere.extremescale.admin.doc/rxsdepschema.html>

When running WebSphere eXtreme Scale grid containers in a stand-alone environment, these two XML configuration files are passed as command-line parameters when you start each container Java virtual machine (JVM) by using **startOgServer**. Because you pass the file names, the files can be named anything that you want. When running WebSphere eXtreme Scale grid containers in a WebSphere Application Server environment, the actual WebSphere Application Server run time inspects the Java Platform, Enterprise Edition (Java EE) modules that are started for the presence of these configuration files. More specifically, it looks for the following files in the META-INF directory of the Java EE module:

- `objectGrid.xml`
- `objectGridDeployment.xml`

The grid containers initialize the object grids that are defined in the configuration files upon start-up of the Java EE application that contains the module. This method makes it easier to manage the WebSphere eXtreme Scale run time without relying on scripts to launch the grid containers. For embedded environments, if you want the configuration files to be external to your modules and .ear or .war files, you can use a technique where you refer to the two configuration files by defining two URLs each with a well-known Java Naming and Directory Interface (JNDI) name. This way, the files remain outside any compiled unit and can be more easily edited by operations as the need arises.

Download material: For an example of WebSphere eXtreme Scale code that allows you to start a WebSphere Application Server application server as an WebSphere eXtreme Scale container using your XML files but with those files external to any compiled .ear or .war file, see the `WAS+XS_ExternalXML_Package.zip` file in the additional materials that are available for this book. For information about downloading this material, see Appendix B, “Additional material” on page 197.

Configuration alternatives: Another alternative for configuring a grid is to invoke the WebSphere eXtreme Scale application programming interface (API). Then, the actual Java or Java EE application contains the configuration, and any modification of the configuration requires you to repackage and redeploy the application. Note, however, that there might be cases where using the API to configure a grid is desirable, for example when defining a temporary grid, or if you want to hide the configuration from users, using the WebSphere eXtreme Scale API can be a better approach.

We do not provide details about how to use the API to configure a grid. You can obtain more information about how to use the API to configure a grid at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.e xtremescale.prog.doc/cxsprgrm.html>

5.1.2 Client XML configuration

An WebSphere eXtreme Scale client also has an ObjectGrid descriptor configuration. The descriptor initially is the same as the descriptor XML that you gave to the containers to which the client connects. The client run time is passed this descriptor configuration transparently at the time that it connects and gets the ObjectGrid reference. In certain cases, it might be necessary to override parts of the object grid configuration on the client. For example, a loader plug-in might make sense on the server but the Loader class probably does not even exist on the client. The specifics of evictors might differ on the client from the server. You might want to turn off certain features, such as the near cache.

Overriding plug-ins

The following plug-ins and attributes can be overridden in the client ObjectGrid descriptor XML file:

- ▶ ObjectGrid plug-ins:
 - TransactionCallback plug-in
 - ObjectGridEventListener plug-in
 - Security-related plug-ins
- ▶ BackingMap plug-ins:
 - ObjectTransformer plug-in (for efficient serialization)
 - Evictor plug-in

- MapEventListener plug-in
- numberOfBuckets attribute (which can be used to turn off the near cache)
- ttlEvictorType attribute
- timeToLive attribute

You also can (and might have to) turn off the reference to a loader plug-in from your server configuration.

We describe plug-ins later in this chapter, but for now, know that the client configuration can specify these overrides. You can obtain information about the client ObjectGrid descriptor XML file at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.webSphere.extremescale.admin.doc/cxscliconfig.html>

Typically, a client ObjectGrid descriptor XML file is packaged with the client Java EE application. The application needs to explicitly use this file when connecting to the object grid, as shown in Example 5-1.

Example 5-1 How to code connection to a grid with a client ObjectGrid descriptor XML file

```
URL objectgridxml = Thread.currentThread().getContextClassLoader()
    .getResource("/META-INF/objectGridClient.xml");
String catalogEndpoint = ServerFactory.getServerProperties()
    .getCatalogServiceBootstrap();
ClientClusterContext ccc = ObjectGridManagerFactory
    .getObjectGridManager().connect(catalogEndpoint, null, objectgridxml);
ObjectGrid grid = ObjectGridManagerFactory.getObjectGridManager()
    .getObjectGrid(ccc, "MyGrid");
```

Disabling near cache

A WebSphere eXtreme Scale client can maintain its own near-side cache, which is automatically enabled when using lockStrategy="OPTIMISTIC" or "NONE". However, you can disable this client cache using the client ObjectGrid descriptor XML file. You must set the numberOfBuckets attribute of the BackingMap to zero (0), as shown in Example 5-2.

Example 5-2 Using a client ObjectGrid descriptor XML file to disable the near cache

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="Grid">
            <backingMap name="Test" numberOfBuckets="0" />
        </objectGrid>
    </objectGrids>
</objectGridConfig>
```

Disabling a loader reference on the client

Because a client inherits the configuration of the containers to which it connects, you might find your client throwing a `ClassNotFoundException` for a loader class that you configured on your container servers. You can use a client ObjectGrid descriptor XML file to fix this situation as well. If you recall, you configured your loader in your server XML file using the "**backingMapPlugins**" parameter on a `BackingMap`. In Example 5-2, you see that the `backingMap` clause has no "**backingMapPlugins**" parameter; the absence of this parameter keeps the client from trying to find the loader class and fixes the problem. A similar technique can be used to blank out references to evictor classes and other plug-ins on the client side if they are not appropriate there.

5.1.3 Server properties

There are a number of configuration parameters that are not set using the server XML files. Each WebSphere eXtreme Scale catalog server or container server can be configured through a local configuration file called `objectGridServer.properties`. We will cover a subset of these properties here. You can find a complete description of all the configuration options that are available in the `objectGridServer.properties` file at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.webSphere.extremescale.admin.doc/rxscontprops.html>

Note the following configuration properties for the `objectGridServer.properties` file:

- ▶ `minThreads` and `maxThreads`
Controls the number of threads that are used by the internal thread pool in the run time for built-in evictors and DataGrid API operations.
- ▶ `serverName`
Sets the server name that is used to identify the server.
- ▶ `zoneName`
Sets the name of the zone to which the server belongs.
- ▶ `enableQuorum`
Enables quorum for the catalog service.
- ▶ `heartBeatFrequencyLevel`
Specifies how often heartbeats occur.
- ▶ `securityEnabled`
Enables container server security when set to true. This property must match the `securityEnabled` property that is specified in the `objectGridSecurity.xml` file that is provided to the catalog server.
- ▶ Secure Sockets Layer (SSL) transport layer security properties
Provides a range of configuration properties to configure SSL transport layer security.

Other security-related configurations, such as authentication, are done in the `security.xml` file. See the product documentation for details.

Configuring server properties in WebSphere Application Server environments

WebSphere eXtreme Scale can automatically load the server properties file upon the start of the container or catalog server in WebSphere Application Server. Create a file called `objectGridServer.properties` and place it in a directory that is in the server's class path.

Warning: The `objectGridServer.properties` file will not be loaded if it is within a Java EE application's class path; it has to be in the server's class path.

For example, you can place the file in the `properties` directory of the WebSphere Application Server profile. When you restart the WebSphere Application Server process, you can confirm that the server properties are loaded from the `SystemOut.log`, as shown in Example 5-3 on page 75.

Example 5-3 WebSphere eXtreme Scale server properties are loaded

```
CW0BJ3142I: This WebSphere Application Server is not associated with a WebSphere
eXtreme Scale zone. In order to start the server in a zone, ensure that the
server's node is within a node group whose name begins with the string
ReplicationZone.
CW0BJ0903I: The internal version of WebSphere eXtreme Scale is v4.2.0 (7.1.0.2)
[cf21118.65559].
CW0BJ0913I: Server property files have been loaded:
file:/opt/IBM/WebSphere/AppServer/profiles/sa-w1201nx1-dmgr/properties/objectGridS
erver.properties.
```

Alternatively, use the following procedure to set a JVM custom property that explicitly references a server properties file on the file system:

1. In the WebSphere administrative console, go to the actual WebSphere Application Server process.
2. In the right panel, click **Java and Process Management** → **Process Definition** → **Java Virtual Machine** → **Custom properties**.
3. Click **New** to create a new JVM custom property:
 - a. Enter `objectgrid.server.props` as the name for the custom property.
 - b. Enter `<file>` as the value, where `<file>` is the file name of the server properties file.
 - c. Optionally, provide a description, for example `WebSphere eXtreme Scale server properties file`.
 - d. Click **Apply**.
4. Save the changes to the master configuration.
5. Restart the WebSphere Application Server process for the changes to take effect.

Configuring server properties in stand-alone environments

WebSphere eXtreme Scale can automatically load the server properties file upon the start of the container or catalog server. Create a file called `objectGridServer.properties`, and place it in a directory that is contained by the server's class path. For example, you can place the file in the `properties` directory of the WebSphere eXtreme Scale product (`$WXS_HOME/properties`).

Warning: If you put this well-named file in the current directory, the file is not found *unless* the current directory is in the class path.

Alternatively, specify the server properties file using the **-serverProps** parameter when you run the **startOgServer** script to launch a catalog server or container. Example 5-4 demonstrates how to specify this file for a catalog server.

Example 5-4 Specifying server properties file using -serverProps

```
[wasuser@sa-w120lnx1 bin]$ ./startOgServer.sh sa-120-catalog -serverProps
/tmp/objectGridServer.properties
...
CW0BJ1001I: ObjectGrid Server sa-120-catalog is ready to process requests.
```

5.1.4 Client properties

There are number of configuration parameters that cannot be set using the client ObjectGrid descriptor XML file. You can configure each WebSphere eXtreme Scale client through a local configuration file called `objectGridClient.properties`.

We do not describe all of the configuration options in detail. However, the following examples show several of the configuration parameters for the `objectGridClient.properties` file:

► `preferZones`

Specifies a list of preferred routing zones. Each specified zone is separated by a comma in the following form:

```
preferZones=ZoneA,ZoneB,ZoneC
```

► `requestRetryTimeout`

Specifies how long to retry a request (in milliseconds). Refer to 7.4, “Configuring failure detection” on page 162 for more details.

► `securityEnabled`

Enables WebSphere eXtreme Scale client security. This setting needs to match the `securityEnabled` setting in the WebSphere eXtreme Scale server properties file, `objectGridServer.properties`.

► `SSL transport layer security properties`

A range of configuration properties is available to configure SSL transport layer security.

You can find a complete description of all the configuration options that are available in the `objectGridClient.properties` file at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.webSphere.extremescale.admin.doc/rxscliprops.html>

Configuring client properties

There are three ways to make the client properties available:

► To configure client properties as a well-named file anywhere in the class path, use:

```
objectGridClient.properties
```

Do not place this file in the system current directory, because it is not supported.

- To configure client properties as a system property in either a stand-alone or WebSphere Application Server configuration, use:
`-Dobjectgrid.client.props=file_name`
 The value is the name of a file in the file system (fully qualified or relative to the location of the script); it cannot be a name based on the class path.
- To configure client properties as a programmatic override, use the `ClientClusterContext.getClientProperties` method. With this method, the data in the object is populated with the data from the properties files. You cannot configure security properties with this method.

5.1.5 Externalizing the server XML configuration in WebSphere Application Server

A common pattern when running WebSphere eXtreme Scale in WebSphere Application Server is to deploy Java EE applications that include the required configuration files. When the run time detects the presence of the server XML files in the META-INF directory of a Java EE module, it launches the WebSphere eXtreme Scale container automatically.

Typically, the Java EE modules are deployed on a cluster, which effectively turns the cluster members into WebSphere eXtreme Scale containers. The grid that is hosted by those servers is registered automatically with the default catalog service domain and, therefore, can be accessed immediately from client applications that are running in the same cell. This method brings benefit in terms of integration, but changes in the configuration require a rebuild of the Java EE applications because they contain the XML files.

In this section, we describe an alternative method that allows for a separation of those XML files and the Java EE application. Although not the default, a number of clients use this method already and are very satisfied with it.

Overview

Instead of relying on the run time to launch the WebSphere eXtreme Scale container, we use a simple servlet to launch it. The web module that contains the servlet is deployed as a standard EAR file on a WebSphere Application Server cluster. Figure 5-2 illustrates the steps when we started the WebSphere eXtreme Scale container.

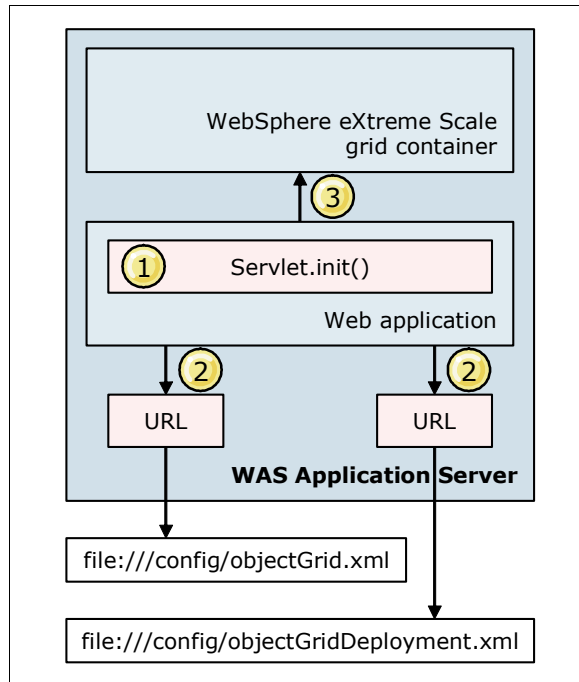


Figure 5-2 Using the servlet life cycle to start the grid container

We used these steps to launch the WebSphere eXtreme Scale container:

1. The servlet is configured to load on start-up, meaning that the servlet's `init()` method is called when the web module is started, which happens when this application .ear is started.
2. The `init()` method looks up two URLs that point to the ObjectGrid descriptor XML file and the deployment policy XML file.
3. The `init()` method then uses the XML files in a WebSphere eXtreme Scale administrative API call to launch the grid container.

Servlet implementation

Example 5-5 shows the implementation of the actual servlet. The `init()` method performs a lookup of the two URLs that refer to the XML files. It then uses those XML files when it initializes the grid container. You can use this code as a starting point for your own implementation. Change the name of the package and the name of the actual servlet class if you want.

Java build path: For the imports to work, `wsobjectgrid.jar` must be on the build path of the integrated development environment (IDE).

Example 5-5 Implementation details of `WXSCContainerServlet`

```

package com.ibm.itso;

import java.net.URL;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import com.ibm.websphere.objectgrid.ObjectGridException;
  
```

```

import com.ibm.websphere.objectgrid.deployment.DeploymentPolicyFactory;
import com.ibm.websphere.objectgrid.server.Server;
import com.ibm.websphere.objectgrid.server.ServerFactory;

public class WXSContainerServlet extends HttpServlet {
    public void init() throws ServletException {
        try {
            System.out.println("DEBUG: WXSContainerServlet init()...");

            // Look up the URLs for the XML configuration files
            InitialContext ctx = new InitialContext();
            URL ogxml = (URL)ctx.lookup("url/objectGridXMLFile");
            URL ogdxml = (URL)ctx.lookup("url/objectGridDeploymentXMLFile");
            System.out.println("DEBUG: Starting a WXS container using the two
                external XML files " + ogxml.getFile() + " and " + ogdxml.getFile() );

            // Get a new WXS container server instance (not initialized yet)
            Server wxsContainerServer = ServerFactory.getInstance();

            // Initialize this server with the XML files obtained from URLs
            com.ibm.websphere.objectgrid.deployment.DeploymentPolicy policy =
                DeploymentPolicyFactory.createDeploymentPolicy(ogdxml, ogxml);
            wxsContainerServer.createContainer(policy);
            System.out.println("DEBUG: A WXS container was started");
        }
        catch (ObjectGridException e) {e.printStackTrace();}
        catch (NamingException e) {e.printStackTrace();}
        return;
    }
}

```

You do not need to do anything in the servlet destroy() method. When you stop the JVM, the whole process, including the WebSphere eXtreme Scale container running in it, stops. No other action is needed, because this approach is a documented and supported way to stop a container JVM. For this reason, there is no WebSphere eXtreme Scale API to stop a container as there is for createContainer().

Enabling Load On Startup in the web module deployment descriptor

The EAR file that contains the web module is deployed on a WebSphere Application Server cluster. By default, all enterprise applications (EAR files) that are deployed on a cluster are started upon the start of the cluster. Thus, our web module is started, but we also need to make sure that the servlet's init() method is called on start. We must enable *Load On Startup* in the web module deployment descriptor.

Figure 5-3 shows the deployment descriptor in Rational® Application Developer 7.5. Note that the option Load On Startup is set to 1.

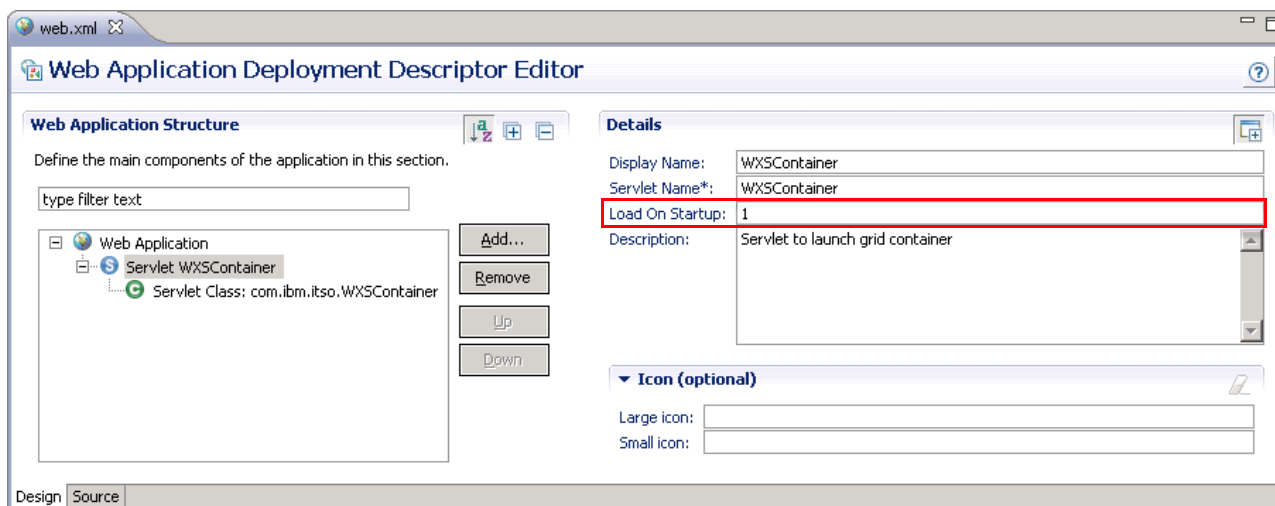


Figure 5-3 Enabling Load On Startup in Rational Application Developer V7.5

The underlying web.xml source for the deployment descriptor reflects this value, as shown in Example 5-6.

Example 5-6 Enabling load-on-startup in the web module deployment descriptor source

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <display-name>WXSContainerModule</display-name>
  <servlet>
    <description>Servlet to launch grid container</description>
    <display-name>WXSContainer</display-name>
    <servlet-name>WXSContainer</servlet-name>
    <servlet-class>com.ibm.itso.WXSContainer</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
</web-app>
```

Defining URLs for XML files

As described earlier in this section, you need to define two URLs in WebSphere Application Server. The URLs point to the XML configuration files for the grid.

XML configuration files: You must ensure that the XML configuration files are accessible from the same location by all of the cluster members. Although not required, the use of a shared file system can ease the administration of these XML files.

Using the Integration Solutions Console, follow these steps to define the URLs:

1. In the left panel, expand **Resources** → **URL**, and click **URLs**.
2. Optional: Set the scope to the cluster where the servlet will be deployed.

3. Now, create the first URL for the ObjectGrid Descriptor XML file:
 - a. Choose **New**.
 - b. Enter objectGridXMLFile for the name of the URL.
 - c. Enter url/objectGridXMLFile for the JNDI name.
 - d. Specify the location of the objectGrid.xml file. For example, enter file:///tmp/wxs/objectGrid.xml if the file is in /tmp/wxs/ on the local file system.
 - e. Optional: Enter a description for the URL.
 - f. Your configuration looks similar to the configuration that is shown in Figure 5-4 on page 81. Click **Apply**.

URLs

☐ Messages

Additional Properties for this object will not be available to edit until its general properties are applied by clicking on either Apply or OK.

URLs > New

Use this page to configure uniform resource locators (URLs), which point to electronically accessible resources, such as a directory file on a machine in a network or a document stored in a database.

Configuration

General Properties

* Scope
cells:sa-w120:clusters:Grid

Provider

Default URL Provider
Create New Provider

* Name
objectGridXMLFile

* JNDI name
url/objectGridXMLFile

* Specification
file:///config/objectGrid.xml

Description
The objectGrid.xml configuration file which defines all Grids, MapSets etc for our WXS containers. Used in combination with the URL "url/objectGridDeploymentXMLfile".

Category

Apply OK Reset Cancel

The additional properties will not be available until the general properties for this item are applied or saved.

Additional Properties

■ Custom properties

Figure 5-4 Creating a URL for the ObjectGrid Descriptor XML file

4. Create the second URL for the Deployment Policy XML file in a similar fashion:
 - a. Choose **New**.
 - b. Enter objectGridDeploymentXMLFile for the name of the URL.
 - c. Enter url/objectGridDeploymentXMLFile for the JNDI name.

- d. Specify the location of the object grid deployment XML file. For example, enter file:///tmp/wxs/objectGridDeployment.xml if the file is in /tmp/wxs/ on the local file system.
 - e. Optional: Enter a description for the URL.
 - f. Click **Apply**.
5. Click **Save** to save the changes to the master configuration.

The configuration now looks similar to the configuration that is shown in Figure 5-5 on page 82.

URLs

Use this page to configure uniform resource locators (URLs), which point to electronically accessible resources, such as a directory file on a machine in a network or a document stored in a database.

☐ Scope: Cell=sa-w120, Cluster=Grid

☒ Show scope selection drop-down list with the all scopes option

Scope specifies the level at which the resource definition is visible. For detailed information on what scope is and how it works, [see the scope settings help](#).

Cluster=Grid

Preferences

New Delete

Select	Name	JNDI name	Scope	Provider	Description	Category
<input type="checkbox"/>	objectGridDeploymentXMLFile	url/objectGridDeploymentXMLFile	Cluster=Grid	Default URL Provider	The objectGridDeployment.xml configuration file which defines all Grids, MapSets etc for our WXS container servers. Used in combination with the URL "url/objectGridXMLFile".	
<input type="checkbox"/>	objectGridXMLFile	url/objectGridXMLFile	Cluster=Grid	Default URL Provider	The objectGrid.xml configuration file which defines all Grids, MapSets etc for our WXS containers. Used in combination with the URL "url/objectGridDeploymentXMLFile".	

Total 2

Figure 5-5 Configured URLs that point to the XML configuration files

Testing the solution

To test the solution, we assume that the URLs are configured and that the EAR application containing the servlet is deployed onto a cluster. We also assume that there is a catalog server running in the deployment manager or that a catalog service domain is running.

When you start the cluster, the grid container in each cluster member initializes automatically. You can confirm this initialization by looking at the SystemOut.log of the WebSphere Application Server cluster member process. Example 5-7 shows a successful start of the ObjectGrid server container.

Example 5-7 Successful start-up of the ObjectGrid server container

```
ApplicationMg A WSVR0200I: Starting application: WXS
ApplicationMg A WSVR0204I: Application: WXS Application build level: Unknown
webapp I com.ibm.ws.webcontainer.webapp.WebGroupImpl WebGroup SRVE0169I:
Loading Web Module: WXSContainerModule.
WASSessionCor I SessionContextRegistry getSessionContext SESN0176I: Will create a
```

```

new session context for application key default_host/WXSWeb
SystemOut      0 DEBUG: WXSContainerServlet init()...
SystemOut      0 DEBUG: Starting a WXS container using the two external XML files
                  /tmp/wxs/objectGrid.xml and /tmp/wxs/objectGridDeployment.xml
ServerImpl     I  CWOBJ2501I: Launching ObjectGrid server
                  sa-w120\sa-w1201nx2\sa-w1201nx2_container1.
PeerManagerSe  I  CWOBJ7700I: Peer Manager service started successfully in server
                  (com.ibm.ws.objectgrid.leader.PeerManager@316d316d) with core
                  group (DefaultCoreGroup).
PeerManager    I  CWOBJ8601I: PeerManager found peers of size 7
ServerImpl     I  CWOBJ8000I: Registration is successful with zone (DefaultZone)
                  and coregroup of (sa-w120_sa-w120DefaultCoreGroup).
ServerImpl     I  CWOBJ1001I: ObjectGrid Server
                  sa-w120\sa-w1201nx2\sa-w1201nx2_container1 is ready to process
                  requests.
XmlObjectGrid  I  CWOBJ4701I: Template map info_server.* is configured in
                  ObjectGrid IBM_SYSTEM_xsastats.server.
SystemOut      0 DEBUG: A WXS container was started

```

5.1.6 Duplicate server names in WebSphere Application Server

It is common for a WebSphere Application Server environment to have a number of processes in the cell that have the same name. For example, the node agent on each node is called the *nodeagent*. Duplicate names are not a problem, because WebSphere Application Server uses a more precise name to identify each process in the cell. So, a node agent is referred to as *cell_name\node_name\nodeagent*.

However, when running WebSphere eXtreme Scale in WebSphere Application Server, these duplicate names can cause issues because WebSphere eXtreme Scale only takes the name of the process into account (that is, *nodeagent*). When the catalog servers or container servers are deployed on processes that have duplicate names, the duplicate names can lead to instability and abnormal behavior. A major concern here is that nothing usually appears to malfunction immediately.

Sometimes, it is not possible or not desirable to avoid duplicate names. Setting the JVM custom property on those JVMs that have duplicate names will ensure that having duplicate names does not present a problem for WebSphere eXtreme Scale.

```
com.ibm.websphere.orb.uniqueServerName=true
```

When using the administrative console in WebSphere Application Server V7.0, you can set this JVM custom property under **Java and Process Management** → **Process Definition** → **Java Virtual Machine** → **Custom properties**.

This process is also documented at this website:

<http://www-01.ibm.com/support/docview.wss?uid=swg21426334>

5.2 Catalog service domain

When running WebSphere eXtreme Scale in a WebSphere Application Server environment, the deployment manager will host a catalog service by default. This function is convenient, but it does not provide any fault tolerance. It is recommended to set up a *catalog service domain* that consists of three catalog servers for production environments. A catalog service

domain is simply a group of catalog servers to ensure the high availability of the catalog service.

5.2.1 Configuring a catalog service domain in a WebSphere environment

To demonstrate how to configure a catalog service running on WebSphere Application Server, we create a catalog service domain that contains the deployment manager and two node agents.

Preferred practice: Our use of the deployment manager and node agents was only for illustration purposes. Typically, you create an application server cluster of three application servers and configure them to be the catalog servers. This way, the catalog server JVMs have no other function than to service the grid and its clients and there is no stealing of JVM resources by other functions (even as simple a function as being a node agent).

The use of the deployment manager node as a catalog server is an anti-pattern for several reasons:

- ▶ Deployment managers are often stopped with no harm; catalog servers must not be stopped.
- ▶ The deployment manager also has a lot of other function that can interfere with the catalog server when it is under stress.
- ▶ You do not want anything else (like a catalog server under stress) to kill your deployment manager right when you need it the most, for example to start up other servers to help with the load.

When configuring the catalog service domain, we obviously define which processes act as a catalog server. However at the same time, we provide a convenient mechanism for clients to bootstrap to the (default) catalog service domain. A WebSphere eXtreme Scale client application deployed in the same cell does not have to provide a list of catalog servers. Instead, it picks this list up from the default catalog service domain. Figure 5-6 on page 85 shows a new catalog service domain configuration for our example.

Catalog service domains

Catalog service domains > New

A WebSphere eXtreme Scale catalog service domain is a highly available collection of catalog servers. This collection of catalog servers can run in WebSphere Application Server server processes within a single cell and core group. The catalog service domain can also define a group of remote servers that run in different WebSphere Application Server cells or as Java SE processes. The catalog service controls the placement of shards and discovers and monitors the health of the container servers in the data grid.

General Properties

* Name

sa-w120

☒ Enable this catalog service domain as the default unless another catalog service domain is explicitly specified.

Catalog Servers

New Delete

Select	Catalog Server Endpoint	Client Port	Listener Port
<input type="checkbox"/>	<input checked="" type="radio"/> Existing application server <div>sa-w120\sa-w120lnx1-dmgr\dmgr</div> <input type="radio"/> Remote server 	6600	
<input type="checkbox"/>	<input checked="" type="radio"/> Existing application server <div>sa-w120\sa-w120lnx2\nodeagent</div> <input type="radio"/> Remote server 	6600	
<input type="checkbox"/>	<input checked="" type="radio"/> Existing application server <div>sa-w120\sa-w120lnx3\nodeagent</div> <input type="radio"/> Remote server 	6600	

Apply

OK

Reset

Cancel

The additional properties will not be available until the general properties for this item are applied or saved.

Additional Properties

Client security properties

Custom properties

Figure 5-6 Creating a catalog service domain

To create this catalog service domain from the WebSphere administrative console, use the following procedure:

1. In the left panel, expand **System Administration** → **WebSphere eXtreme Scale** → **Catalog service domains**.
2. In the right panel, click **New**.
3. Enter a name for the catalog server domain, for example, sa-w120.
4. Select **Enable this catalog service domain as the default**.
5. Now, add the three WebSphere processes to the catalog service domain:
 - a. To add the deployment manager, select the deployment manager in the **Existing application server** drop-down menu, and specify a TCP port that is not in use for the client port, for example, 6600.
 - b. To add each of the node agents, click **New**. Then, select a node agent from the list in the **Existing application server** drop-down menu. Specify a TCP port that is not in use for the client port, for example, 6600.
6. The configuration now looks similar to the configuration that is shown in Figure 5-6. Click **Apply**.

7. Click **Save** to save the changes to the master configuration.

Java Management Extensions (JMX) ports: Prior to WebSphere eXtreme Scale 7.1.0.2, you also had to specify the listener and JMX ports for each WebSphere process that was added to the catalog service domain. This specification is no longer required, because the ports are now picked up automatically from the WebSphere configuration.

After creating (or changing) the catalog service domain configuration, you need to restart all processes that are involved. In this example, we have to stop the deployment manager and both node agents. Also, note that you need to start all catalog servers from the domain together, because these servers communicate with each other in a peer-to-peer fashion to establish who will host the master catalog service.

After restarting, you can verify the status of the catalog service domain in the Integrations Solutions Console, as shown in Figure 5-7. You can also test the connection here.

Catalog service domains ?

[Catalog service domains](#) > **sa-w120**

A WebSphere eXtreme Scale catalog service domain is a highly available collection of catalog servers. This collection of catalog servers can run in WebSphere Application Server server processes within a single cell and core group. The catalog service domain can also define a group of remote servers that run in different WebSphere Application Server cells or as Java SE processes. The catalog service controls the placement of shards and discovers and monitors the health of the container servers in the data grid.

General Properties

* Name

☒ Enable this catalog service domain as the default unless another catalog service domain is explicitly specified.

Additional Properties

- ☐ [Client security properties](#)
- ☐ [Custom properties](#)

Catalog Servers

Select	Catalog Server Endpoint	Client Port	Listener Port	Status
<input type="checkbox"/>	sa-w120\sa-w120\inx1-dmq\dmqr	6600		Up
<input type="checkbox"/>	sa-w120\sa-w120\inx2\nodeagent	6600		Up
<input type="checkbox"/>	sa-w120\sa-w120\inx3\nodeagent	6600		Up

Figure 5-7 Examining the status of the catalog service domain

You can also use the SystemOut.log of the deployment manager and the two node agents to confirm that they are indeed running a catalog service. Example 5-8 shows the SystemOut.log message.

Example 5-8 Log message to indicate that the catalog server is starting

```
[10/21/10 14:24:52:296 CDT] 0000000c ServerImpl I CW0BJ2518I: Launching  
ObjectGrid catalog service: thinkCell01\thinkNode02\nodeagent.
```

When all three catalog servers are running, they establish a catalog service cluster where one catalog server is elected to act as the master catalog service. The remaining catalog servers will act as standby servers. The message from the SystemOut.log, which is shown in Example 5-9, indicates that this catalog server acts as the master.

Example 5-9 Log message to provide the status of the catalog server cluster

```
[10/21/10 14:25:37:343 CDT] 00000012 CatalogServer I   CW0BJ8109I: Updated catalog
service cluster CatalogCluster[thinkCell01, 1 master: 1 standbys] from server
thinkCell01\thinkNode02\nodeagent with entry CatalogServerEntry...
```

Important: Because the deployment manager now also acts as a catalog server, it has become a more important part of the runtime infrastructure. Prior to use with WebSphere eXtreme Scale, stopping and starting the deployment manager did not impact the availability of applications running in the cell. However, stopping and starting the deployment manager *does* have an impact when the deployment manager is running a catalog service. Stopping the deployment manager reduces the number of running catalog servers. When quorum is enabled, the catalog service loses quorum, and no master catalog server is active.

5.3 ObjectGrid plug-ins

A ObjectGrid plug-in is a plug-in that affects the behavior of a grid.

There are multiple ObjectGrid plug-ins:

- ▶ TransactionCallback: Provides transaction life-cycle events
- ▶ ObjectGridEventListener: Provides ObjectGrid life-cycle events for the ObjectGrid, shards, and transactions
- ▶ SubjectSource, ObjectGridAuthorization, and SubjectValidation: Custom authentication and authorization mechanisms
- ▶ MapAuthorization (deprecated): Custom authorization mechanism

This section describes the TransactionCallback and ObjectGridEventListener plug-ins.

5.3.1 The TransactionCallback plug-in

Create, read, update, and delete operations in a map happen in the context of an ObjectGrid transaction. When the ObjectGrid interface is used to inline cache a database, the grid is usually configured to use a loader plug-in. The loader plug-in is responsible for fetching data from a back-end database to put into a WebSphere eXtreme Scale map or for reflecting the changes that occur in a map into the back-end database.

A TransactionCallback plug-in allows the WebSphere eXtreme Scale transactions to be coordinated with the transactions of the back-end system in the case where your application accepts WebSphere eXtreme Scale as the transaction coordinator (meaning that your application code explicitly calls the Session.begin(), Session.getMap(), and Session.commit() methods). A TransactionCallback plug-in manages the begin and commit transaction life-cycle events. WebSphere eXtreme Scale does not support two-phase commit transactions (XA), but it has been reported that a WebSphere eXtreme Scale transaction can be used successfully in a global transaction if it is the last participant (which is supported in

WebSphere Application Server). For this approach to work, you must write a WebSphere eXtreme Scale Resource Adapter, which is not currently part of the product.

A TransactionCallback plug-in can be used both on the client and on the container servers. On the client, it is called as part of the client-side WebSphere eXtreme Scale transaction processing (begin, commit, rollback, and so on). On the container, it is called similarly as part of the container transaction, which is commonly begun and committed in coordination with the client transaction's commit phase. In the case of a write-behind loader, the container transaction is begun and committed periodically based on your write-behind configuration. One curious point (which might be fixed at a future time) is that, on the container side, your TransactionCallback begin() method will not be reliably called as it is on the client side. This point is usually not a serious problem; to compensate, do nothing in begin() and instead put the logic to get the necessary state and begin your back-end system transaction as a singleton operation within the loader get() and batchUpdate() methods. By "singleton", we mean to use "if" logic to insure that you only begin the back-end transaction once even if your Loader.get() or batchUpdate() is called several times during one WebSphere eXtreme Scale transaction.

A loader cannot be associated with more than one TransactionCallback plug-in, but a TransactionCallback plug-in can be associated to more than one loader. TransactionCallback and loader plug-ins can share transactional information, such as a connection to a database, using plug-in slots. There are two built-in TransactionCallback plug-ins with WebSphere eXtreme Scale:

- ▶ The JPATxCallback plug-in
- ▶ The WebSphereTransactionCallback plug-in

The JPATxCallback plug-in

The JPATxCallback plug-in is designed to be used in conjunction with the built-in Java Persistence API (JPA) loader plug-in. The JPATxCallback plug-in manages the transactions against the JPA back end automatically. To use the JPATxCallback plug-in, add the following elements in the ObjectGrid descriptor XML file:

- ▶ In the objectGrid element, add a bean element with the following settings:
 - The ID attribute is set to TransactionCallback.
 - The className attribute is set to the class name `com.ibm.websphere.objectgrid.jpa.JPATxCallback`.
- ▶ In the TransactionCallback bean element, add the required property element with the following settings:
 - The name attribute is set to `persistenceUnitName`.
 - The type attribute is set to `java.lang.String`.
 - The value attribute is set to the persistence unit name. This name must correspond to a persistent unit name given in the persistence XML file. This file is not part of the WebSphere eXtreme Scale configuration. Refer to the JPA documentation to learn more about configuring a persistent unit.

Example 5-10 shows an ObjectGrid descriptor XML file using the JPATxCallback plug-in with the JPA loader plug-in.

Example 5-10 Example of using the JPATxCallback in an ObjectGrid descriptor XML file

```
<?xml version="1.0" encoding="utf-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
```

```

<objectGrids>
  <objectGrid name="UserGrid" txTimeout="60">
    <bean id="TransactionCallback"
      className="com.ibm.websphere.objectgrid.jpa.JPATxCallback">
      <property name="persistenceUnitName" type="java.lang.String"
        value="userPUDB2DS" />
    </bean>
    <backingMap name="User" pluginCollectionRef="User"
      lockStrategy="OPTIMISTIC" />
  </objectGrid>
</objectGrids>
<backingMapPluginCollections>
  <backingMapPluginCollection id="User">
    <bean id="Loader" className="com.ibm.websphere.objectgrid.jpa.JPALoader">
      <property name="entityClassName" type="java.lang.String"
        value="com.ibm.websphere.sample.xs.inlinebuffer.model.User" />
    </bean>
  </backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>

```

The WebSphereTransactionCallback plug-in

The WebSphereTransactionCallback plug-in is designed to be used in enterprise beans running in WebSphere Application Server. With this plug-in, WebSphere eXtreme Scale transactions are treated as container-managed transactions and, as such, there is no need to explicitly call the `Session.begin()` and `Session.commit()` methods. The Enterprise JavaBeans (EJB) container transaction manager uses the method boundaries for the transaction demarcations.

To use the WebSphereTransactionCallback plug-in, add a bean element to the ObjectGrid descriptor XML file with the following settings:

- ▶ The ID attribute is set to `TransactionCallback`.
- ▶ The `className` attribute is set to the class name `com.ibm.websphere.objectgrid.plugins.builtins.WebSphereTransactionCallback`.

Example 5-11 shows an ObjectGrid descriptor XML file using the WebSphereTransactionCallback plug-in.

Example 5-11 Example of using the WebSphereCallback in an ObjectGrid descriptor XML file

```

<?xml version="1.0" encoding="utf-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="UserGrid" txTimeout="60">
      <bean id="TransactionCallback"
        className="com.ibm.websphere.objectgrid.plugins.builtins.WebSphereTransactionCallback">
      </bean>
    </objectGrid>
  </objectGrids>
</objectGridConfig>

```

The custom TransactionCallback plug-in

If you want to use your own implementation of a loader, you might need to write a paired TransactionCallback plug-in. Describing how to write a TransactionCallback plug-in is beyond the scope of this chapter, but note that writing the plug-in consists of implementing the five methods of the `com.ibm.websphere.objectgrid.plugins.TransactionCallback` interface:

- ▶ `begin()`
- ▶ `commit()`
- ▶ `initialize()`
- ▶ `rollback()`
- ▶ `isExternalTransactionActive()`

To use a TransactionCallback plug-in, add the following elements in the ObjectGrid descriptor XML file:

- ▶ In the `objectGrid` element, add a bean element with the following settings:
 - The `ID` attribute is set to `TransactionCallback`.
 - The `className` attribute is set with the class name of your TransactionCallback implementation.
- ▶ In the TransactionCallback bean element, add any required properties, depending on the plug-in implementation. As for any custom plug-in for WebSphere eXtreme Scale, you can add a property by simply adding an attribute to your plug-in implementation in the style of JavaBeans (you must include both a getter and a setter method even though you might not intend to ever use the getter. WebSphere eXtreme Scale requires that the getter method is there before WebSphere eXtreme Scale will recognize the property). In Example 5-12, the TransactionCallback class `com.ibm.websphere.objectgrid.jpa.JPATxCallback` is presumed to have an attribute `private String persistenceUnitName` with methods `getPersistenceUnitName()` and `setPersistenceUnitName(String name)`.

Example 5-12 Configuration of TxCallback and loader plug-ins

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="UserGrid" txTimeout="60">
      <bean id="TransactionCallback"
        className="com.ibm.websphere.objectgrid.jpa.JPATxCallback">
        <property name="persistenceUnitName" type="java.lang.String"
          value="userPUDB2DS"/>
        </bean>
        <backingMap name="User" pluginCollectionRef="User"
          lockStrategy="PESSIMISTIC" writeBehind="T60;C500"/>
      </objectGrid>
    </objectGrids>
    <backingMapPluginCollections>
      <backingMapPluginCollection id="User" >
        <bean id="Loader" className="com.ibm.websphere.objectgrid.jpa.JPALoader">
          <property name="entityClassName" type="java.lang.String"
            value="com.ibm.websphere.sample.xs.inlinebuffer.model.User"/>
          </bean>
        </backingMapPluginCollection>
      </backingMapPluginCollections>
    </objectGrids>
  </objectGridConfig>
```

```
</backingMapPluginCollections>  
</objectGridConfig>
```

5.3.2 The ObjectGridEventListener plug-in

An ObjectGridEventListener plug-in provides WebSphere eXtreme Scale ObjectGrid shard and transaction life-cycle events. These events include ObjectGrid shard initialization, destroying, activation and deactivation, and transaction beginning and ending. WebSphere eXtreme Scale provides two built-in implementations of the ObjectGridEventListener plug-in:

- ▶ JMSObjectGridListener
- ▶ TranPropListener

It is also possible to implement a custom version of this plug-in.

The JMSObjectGridEventListener plug-in

The JMSObjectGridEventListener is a Java Message Service (JMS) implementation of the ObjectGridEventListener interface. You can use it for two purposes:

- ▶ Near cache invalidation
- ▶ Peer-to-peer replication

Peer-to-peer replication warning: Using the JMSObjectGridEventListener plug-in for peer-to-peer replication is no longer recognized as a preferred practice. The plug-in can be difficult to set up and does not scale well. The preferred practice for peer-to-peer replication is to use the WebSphere eXtreme Scale multi-master replication feature.

Near cache invalidation enables WebSphere eXtreme Scale client applications to keep their near cache content synchronized with each other or with the content of the WebSphere eXtreme Scale server cache when another client is updating the server cache. You can use peer-to-peer replication to maintain the same data between two separate distributed or local WebSphere eXtreme Scale servers.

Using the built-in JMSObjectGridEventListener plug-in is the preferred method when a client application needs to be notified of updates that occur in the server grid. Notifications are processed in the onMessage() method implementation. This method gives the client the opportunity to update its near cache. It also requires you to configure JMS and Java Naming and Directory Interface (JNDI) information. If you are running WebSphere Application Server, you can use its default messaging provider. Otherwise, you need a JMS provider, such as WebSphere MQ. Because of the asynchronous nature of JMS, notifications of updates in the near cache occur after a certain delay.

Important: The built-in JMSObjectGridEventListener does nothing. It is designed to be extended, and the user is in charge of writing an invalidation mechanism in the onMessage() method.

To use near cache invalidation, you need to complete the following tasks:

1. Write a JMSObjectGridEventListener extension that implements the onMessage() method. You can find an example of such an implementation in the WebSphere eXtreme Scale Information Center:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.e xtremescale.admin.doc/cxsjmsevtls.html>

2. Configure the JMS topic and the related JNDI names. We do not explain this task here, but, remember, when designing the JMS topic connection factory, that each container server in a publisher role creates one connection for each partition during the `JMSObjectGridEventListener` initialization. You must size the connection pool size according to your topology.

In addition to these tasks, you need to configure the `JMSObjectGridEventListener` plug-in by adding the following elements in the `ObjectGrid` descriptor XML file:

- ▶ In the `objectGrid` element, add a bean element with the following settings:
 - The `ID` attribute is set to `JMSObjectGridEventListener`.
 - The `className` attribute is set with the class name of your `JMSObjectGridEventListener` extensions.
- ▶ In the `JMSObjectGridEventListener` bean element, add the following required property elements:
 - A property element with the following settings:
 - The `name` attribute is set to `invalidationModel`.
 - The `type` attribute is set to `java.lang.String`.
 - The `value` attribute is set to `NONE_INVALIDATION_MODEL`, `CLIENT_SERVER_MODEL`, or `CLIENT_AS_DUAL_ROLES_MODEL`.
 - A property element with the following settings:
 - The `name` attribute is set to `invalidationStrategy`.
 - The `type` attribute is set to `java.lang.String`.
 - The `value` attribute is set to `INVALIDATE`, `INVALIDATE_CONDITIONAL`, `PUSH`, `PUSH_CONDITIONAL`, `PUSH_INCLUDED`, or `PUSH_INCLUDED_CONDITIONAL`.
 - A property element with the following settings:
 - The `name` attribute is set to `jms_topicConnectionFactoryJndiName`.
 - The `type` attribute is set to `java.lang.String`.
 - The `value` attribute is set to the JNDI name of the Topic connection factory.
 - A property element with the following settings:
 - The `name` attribute is set to `jms_topicJndiName`.
 - The `type` attribute is set to `java.lang.String`.
 - The `value` attribute is set to the JNDI name of the Topic for publishing or receiving notifications.
 - A property element with the following settings:
 - The `name` attribute is set to `jms_topicName`.
 - The `type` attribute is set to `java.lang.String`.
 - The `value` attribute is set to the name of the Topic for publishing or receiving notifications.

Optionally, If security is enabled to connect to the topic, you can add the following property elements:

- ▶ A property element with the following settings:
 - The `name` attribute is set to `jms_userid`.
 - The `type` attribute is set to `java.lang.String`.

- The value attribute is set to the name of a user authorized to connect to the topic.
- ▶ A property element with the following settings:
 - The name attribute is set to `jms_password`.
 - The type attribute is set to `java.lang.String`.
 - The value attribute is set to the password corresponding to the user that was set in the previous property.

There are other optional properties that you can use to customize the plug-in. For a complete description of all properties, refer to this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extramescale.javadoc.doc/topics/com/ibm/websphere/objectgrid/plugins/builtins/JMSObjectGridEventListener.html>

Example 5-13 shows the configuration of near cache invalidation.

Example 5-13 Extract of near cache invalidation

```
<bean id="ObjectGridEventListener"
  className="com.ibm.websphere.sample.xs.inlinebuffer.plugins.ExtendedJMSObjectGridEventListener">
  <property name="invalidationModel" type="java.lang.String"
    value="CLIENT_SERVER_MODEL" description="" />
  <property name="invalidationStrategy" type="java.lang.String"
    value="INVALIDATE" description="" />
  <property name="jms_topicConnectionFactoryJndiName" type="java.lang.String"
    value="jms/WXSMapInvalidationTCF" description="" />
  <property name="jms_topicJndiName" type="java.lang.String"
    value="jms/WXSMapInvalidationTopic" description="" />
  <property name="jms_topicName" type="java.lang.String"
    value="WXSMapInvalidationTopic" description="" />
  <property name="enableOnServerObjectGrid" type="boolean" value="true"
    description="" />
  <property name="enableOnClientObjectGrid" type="boolean" value="true"
    description="" />
  <property name="jms_userid" type="java.lang.String" value="wasuser"
    description="" />
  <property name="jms_password" type="java.lang.String" value="password"
    description="" />
</bean>
```

The TranPropListener plug-in

The TranPropListener plug-in is another way to operate peer-to-peer replication. This plug-in must run in a WebSphere Application Server environment with the high availability (HA) manager active. It is not considered a preferred practice, and it might be deprecated in the future.

A custom plug-in

You can write an `ObjectGridEventListener` and configure its use in the `ObjectGrid` descriptor XML file. To plug in an `ObjectGridEventListener`, add a bean element with the ID `ObjectGridEventListener` and the class name of the plug-in implementation in the `objectGrid` element. Example 5-14 provides an example.

Example 5-14 Plug in an ObjectGridEventListener

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
  <objectGrid name="grid1">
    <bean id="ObjectGridEventListener"
      className="com.company.org.CustomObjectGridEventListener" />
    <backingMap name="map1" ttlEvictorType="NONE" />
  </objectGrid>
</objectGrids>
```

Depending on which events the custom listener must handle, it can implement one or more of the interfaces that are listed in Table 5-1.

Table 5-1 List of interfaces and their methods

Class name	Method to implement	Description
com.ibm.websphere.objectgrid.plugins.ObjectGridEventGroup.ShardLifecycle	<ul style="list-style-type: none">▶ initialize()▶ destroy()	<ul style="list-style-type: none">▶ ObjectGrid shard initialization.▶ ObjectGrid shard destroy.
com.ibm.websphere.objectgrid.plugins.ObjectGridEventGroup.ShardEvents	<ul style="list-style-type: none">▶ shardActivated()▶ shardDeactivated()	<ul style="list-style-type: none">▶ Called when the shard becomes a primary.▶ Called when the shard becomes a replica.
com.ibm.websphere.objectgrid.plugins.ObjectGridEventGroup.TransactionEvents	<ul style="list-style-type: none">▶ transactionBegin()▶ transactionEnd()	<ul style="list-style-type: none">▶ Called at the beginning of a session transaction.▶ Called at the end of a transaction.

Note: Implementing the com.ibm.websphere.objectgrid.plugins.ObjectGridEventListener is the same as implementing the following interfaces:

- ▶ com.ibm.websphere.objectgrid.plugins.ObjectGridEventGroup.TransactionEvents
- ▶ com.ibm.websphere.objectgrid.plugins.ObjectGridEventGroup.ShardLifecycle

5.4 BackingMap plug-ins

A BackingMap plug-in is a plug-in that affects the behavior of a map. The following plug-ins are available for a BackingMap:

- ▶ Evictor: An evictor plug-in is a mechanism that is provided for evicting cache entries. There are default evictors and an interface for creating custom evictors.
- ▶ ObjectTransformer: An ObjectTransformer plug-in allows you to serialize, deserialize, and copy objects in the cache, either alone or in conjunction with the object's own Serializable or Externalizable methods.
- ▶ OptimisticCallback: An OptimisticCallback plug-in allows you to customize versioning and comparison operations of cache objects when you are using the optimistic lock strategy.

- ▶ **MapEventListener:** A MapEventListener plug-in provides callback notifications and significant cache state changes that occur for a BackingMap.
- ▶ **Indexing:** Use the indexing feature, which is represented by the MapIndex plug-in, to build an index or several indexes on a BackingMap to support non-key data access.
- ▶ **Loader:** A loader plug-in on an ObjectGrid map acts as a layer of code between a BackingMap and data that is typically kept in a persistent store on either the same system or another system. (Server side only)

This section will look at loader, OptimisticCallback, MapEventListener, and indexing plug-ins. For information about the ObjectTransformer plug-in, see 6.4, “Serialization performance” on page 127. For information about evictors, see 6.2, “Evictor performance preferred practices” on page 115.

5.4.1 Loaders: Choices and configurations

A *loader* is a plug-in that acts between a WebSphere eXtreme Scale backing map and a back-end system (typically, a database). When configured, a loader plug-in can automatically fetch data from the back-end system when it is not available in the map (cache miss) after a client has called the ObjectMap.get() method. A loader can also update data in the back-end system after modifications in the map have been committed. Finally, a loader plug-in can preload data from the back-end system into a map. However, using the preload method is not recognized as a preferred practice for most scenarios because of how and when it is called. A Java client, perhaps in combination with a MapGridAgent, is usually a better design; see “Using a custom loader plug-in” on page 99.

When using loaders in a backing map, consider the following facts:

- ▶ A backing map can have only one loader plugged in.
- ▶ The same loader can be plugged in to more than one backing map.
- ▶ A near cache cannot have a loader.

There are two types of loader plug-ins:

- ▶ JPA loader
- ▶ Custom loader

Using a JPA loader plug-in usually requires the use of the JPATxCallback plug-in to ensure that ObjectGrid can coordinate its transactions with the back-end transaction manager. (Note that “write-behind” is not a type of loader; it is a configuration choice on when and how a loader is invoked.)

Using the JPA loader plug-in

A JPA loader plug-in uses the Java Persistence API (JPA) to access data through a JPA implementation. WebSphere eXtreme Scale does not implement JPA, but the JPA loader plug-in can be used with at least two JPA implementations:

- ▶ Hibernate
- ▶ OpenJPA

These two JPA implementations support a majority of database management, but it is worth verifying that they support your database management system (DBMS) specifications. WebSphere eXtreme Scale supplies the following built-in JPA loaders:

- ▶ The JPA loader plug-in
- ▶ The JPAEntityLoader plug-in (not described in this book)

Using the built-in JPA loader plug-in is just a matter of configuration. Typically, it is not necessary to write any JPA code. Using the JPA loader (or JPAEntityLoader) implies that your application is written to directly access a grid using WebSphere eXtreme Scale APIs. The loader uses JPA to interact with your chosen database. This approach differs from the WebSphere eXtreme Scale L2 cache, where your application is written to use JPA (or Hibernate) APIs and WebSphere eXtreme Scale acts as a cache engine behind these APIs.

Figure 5-8 illustrates the architecture of a WebSphere eXtreme Scale backing map using a JPA loader plug-in.

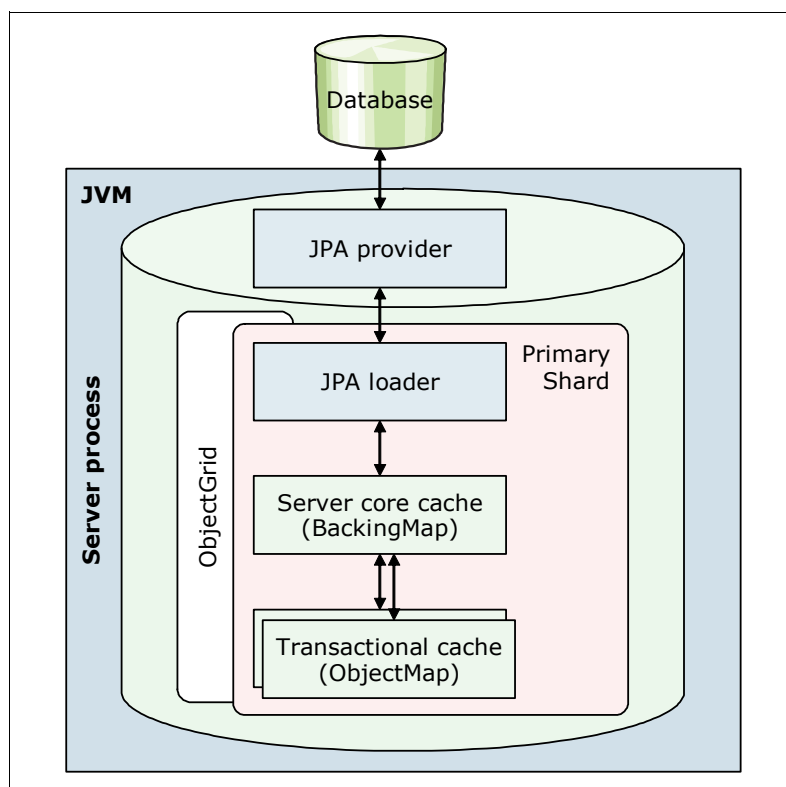


Figure 5-8 Architecture of a WebSphere eXtreme Scale backing map using a JPA loader plug-in

To configure the JPA loader plug-in, add the following elements in the ObjectGrid descriptor XML file:

- ▶ In the objectGrid element, add a bean element with the following settings:
 - The ID attribute is set to TransactionCallback.
 - The className attribute is set to com.ibm.websphere.objectgrid.jpa.JPATxCallback.
- ▶ In the bean element, add a property element with the following settings:
 - The name attribute is set to persistenceUnitName.
 - The type attribute is set to java.lang.String.
 - The value attribute is set to a name of your choice.
- ▶ In the objectGrid element and under the bean element, add a backingMap element with the following settings:
 - The name attribute is set to the name of a map. (The map must also be defined in the deployment policy XML file.)
 - The pluginCollectionRef attribute is set to a name of your choice.

- Any other attributes that are required to customize the backing map are set.
- ▶ In the `objectGridConfig` element, add a `backingMapPluginCollections` element.
- ▶ In the `backingMapPluginCollections` element, add a `backingMapPluginCollection` with the `ID` attribute set to the same name that is used in the `pluginCollectionRef` attribute.
- ▶ In the `backingMapPluginCollection` element, add a `bean` element with the following settings:
 - The `ID` attribute is set to `Loader`.
 - The `className` attribute is set to `com.ibm.websphere.objectgrid.jpa.JPALoader`.
- ▶ In the `bean` element, add a `property` element with the following settings:
 - The `name` attribute is set to `entityClassName`.
 - The `type` attribute is set to `java.lang.String`.
 - The `value` attribute is set to the name of the class name of the entity, which must be loaded in the backing map.
- ▶ Repeat the `entityClassName`.

Example 5-15 shows an ObjectGrid descriptor XML file using the JPA loader plug-in.

Example 5-15 Configuration of the built-in JPA loader plug-in

```
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="UserGrid" txTimeout="60">
      <bean id="TransactionCallback"
        className="com.ibm.websphere.objectgrid.jpa.JPATxCallback">
        <property name="persistenceUnitName" type="java.lang.String"
          value="userPUDB2DS"/>
      </bean>
      <backingMap name="User" pluginCollectionRef="User"
        lockStrategy="PESSIMISTIC"/>
    </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
    <backingMapPluginCollection id="User" >
      <bean id="Loader" className="com.ibm.websphere.objectgrid.jpa.JPALoader">
        <property name="entityClassName" type="java.lang.String"
          value="com.ibm.websphere.sample.xs.inlinebuffer.model.User"/>
      </bean>
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

Preloading a map with the JPA loader plug-in (not preferred in most situations)

Preloading a backing map with the JPA loader plug-in is supported by WebSphere eXtreme Scale but, in many circumstances, it is not preferred for the following reasons:

- ▶ Preloading a backing map with the JPA loader plug-in can be a slow process, depending on how the grid is organized to hold the class instances, especially if there are relationships between classes. The queries that are generated by JPA can be expensive, and the database can become a bottleneck quickly. This issue is the case, for example, when an instance of a class has a lot of children, and a `JOIN` clause must be executed.

- ▶ The JPLoader preloader runs in one and only one container server, even in a distributed grid. Preloading a large amount of data can be memory consuming. The container server in which the preloading is running can run out of memory, compromising the availability of the grid.

Preloading a map with a client loader (preferred practice)

The preferred practice to preload a map that has the JPA loader plugged in is to use a client loader. A *client loader* is a WebSphere eXtreme Scale client application that fetches data from the back-end system and inserts it in the grid using the ObjectGrid API. The use of a client loader is the preferred method for preloading maps. A client loader offers the following advantages:

- ▶ Provides better control when preloading database tables by avoiding expensive queries.
- ▶ Depending on your data, it might be possible to execute multiple instances of the same client simultaneously to reduce the preload time.
- ▶ A client loader can run parallel MapGridAgents, which are used to process operations against map entries in a remote ObjectGrid, to reduce the time of putting entries in multi-partitioned maps.
- ▶ A client loader is useful when data is not stored in a database.
- ▶ A client loader can be invoked manually or automatically.

A client loader (*"preloader"* is really a better name than *"loader"* for this sort of client) can be designed and the grid configured so that the preloader runs when the grid is available to it but not to other client applications.

To achieve this situation, we use a combination of XML configuration and Java code in the preloader. We will set the initial state of our grid to "PRELOAD" by adding a parameter to the objectGrid element in our ObjectGrid descriptor XML file:

```
<objectGrid name="MyGrid" txTimeout="15" initialState="PRELOAD">
```

This action causes our grid, when all containers are started, to not immediately move into the "ONLINE" state (which is the default). Instead, it moves into the "PRELOAD" state (messages in the SystemOut.log file show each partition moving into this state). When a grid is in the "PRELOAD" state, applications attempting to access the grid using a normal session will get a "grid unavailable" exception. Our preloader code, however, knows better. Our preloader will connect to our grid as usual and get a reference to our ObjectGrid as usual but will not call "grid.getSession()" to obtain a session from which to get the maps to preload. Instead, our preloader will call the static method:

```
ClientLoaderFactory.getClientLoaderSession(Session session)
```

The session returned from this call is allowed to access a grid that is currently in PRELOAD state. Normal WebSphere eXtreme Scale APIs can be used from this point to preload objects into the grid. When the preloading is complete and you are ready to allow other clients to access the grid, your preloader must call the following API:

```
StateManagerFactory.getStateManager().setObjectGridState(AvailabilityState.ONLINE, grid);
```

This method does not return until each shard in the grid has transitioned to the ONLINE state (or if it times out).

When trying to set up preloading, start your testing with a small number of records to be sure that everything works well.

Using a custom loader plug-in

When it is not possible to use the built-in JPA loader because you want to add custom loader logic or when the back-end system does not support JPA, such as with web services, then you can write and configure a custom loader. The configuration of a custom loader is the same as the JPA loader except for the following conditions:

- ▶ The value of the `className` attribute in the Loader bean element must be the class name of the custom class loader that you write.
- ▶ It is recommended to write and associate a custom transaction callback with the custom loader. We describe the TransactionCallback plug-in in 5.3.1, “The TransactionCallback plug-in” on page 87.

We do not describe here how to write a custom loader, because it is explained well in the WebSphere eXtreme Scale Information Center:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extramescale.prog.doc/cxsloadwrite.html>

Preloading a map with a client (preferred practice)

One point that is not explained well in the information center is the reason why, for most situations, the loader `preloadMap()` method is not a good choice for preloading data into a grid. The core reason is that there is an instance of the Loader class for each partition that is defined in your grid. The `preloadMap()` method will be called once on each of those instances, and the intention is that you will preload only that partition. In most situations, you do not have a way to design a query that will fetch only data intended for one partition. Usually, there is only a single query that will fetch all the data to be preloaded into the entire grid. Sometimes, you can design a set of queries that will partition your data but not necessarily into single-partition sets (for example, you might perform separate queries for each zip code but you probably want many zip codes to be stored into each partition). It is certainly possible to discover your partition number within `preloadMap()` and only perform preloading if you are partition 0. However, there is no particular advantage to this approach, because you do not have especially fast access to other partitions. Writing a Java application that is devoted to preloading data and using the normal WebSphere eXtreme Scale client APIs to write the data you have fetched from your back end gives you more direct control over when this preloading occurs (for example, system administrators can preload again without restarting the containers), and it performs just as well.

There are several design choices for preloaders, which vary in how well they perform. Your choice depends on whether you can partition the data. To understand these design choices, you must understand what we mean, in WebSphere eXtreme Scale terms and to an extent in general terms, when we say “partition the data”. In practical terms, we mean you can perform these tasks:

1. Design a set of N queries where each query returns a unique set of data.
2. The sum of the results of all queries is the complete set of data to be preloaded into the grid.
3. Executing the set of N queries in parallel is significantly faster than executing a single query to retrieve the complete set of data or if there is no single query to execute.

If you can partition your data, typically, the fastest preload technique is to execute multiple independent preload client applications, each using one of the N queries. Within a single client, if there are a large enough number of rows returned (thus number of objects to be preloaded), you might still want to use the Parallel Batch Preload pattern in the clients.

If you cannot partition your data, you will have a single preload client application, which does as much parallel preloading as possible, in a batch manner. Note that another way of saying “you cannot partition your data” is to say “The only way (or the most efficient way

overall) I can retrieve the complete set of data to be preloaded into the grid is by using a single query". This single query can only (profitably) be run in a single preload client application. It is the fact that you must use a single query that limits your ability to do things in parallel; the Parallel Batch Preloader pattern gets you as much parallel preloading as possible given this fact.

The core of the Parallel Batch Preload pattern is the following logic:

- a. Execute a query or otherwise obtain the data for all (or a sizable number) of key/value pairs to be preloaded.
- b. For each key/value pair, call the local WebSphere eXtreme Scale API that returns the partition number where this key (and value) will be stored when it is later inserted. This API is a purely local API and is typically fast, so there is no significant overhead in calling it. Starting with your ObjectGrid object ("grid") and the String name of the BackingMap that you are preloading ("mapName"), you can obtain the partition for a key with the following Java code:

```
int partition = grid.getMap(mapName).getPartitionManager().getPartition(key);
```
- c. Sort the key/value pairs into "buckets" by the partition to which they belong.
- d. As you collect a large enough "bucket" of pairs for a given partition number, use a WebSphere eXtreme Scale MapGridAgent to batch-insert the bucket of pairs to its partition. By using an agent, you achieve many inserts at the cost of only one Remote Procedure Call (RPC).

Using write-behind caching with loader plug-ins

A WebSphere eXtreme Scale loader plug-in can implement code in the `batchUpdate()` method from the `com.ibm.websphere.objectgrid.plugins.Loader` interface. This configuration is the case, for example, for the built-in JPA loader plug-in. When a backing map plugs in such a loader and when a map inserts or deletes, or updates are committed, the `batchUpdate()` method is invoked to reflect the change in the back-end system.

Because this action happens transparently and synchronously, this action is called *write-through caching*. Write-through caching is the default behavior and can, in certain cases, be a problem. Especially, when a lot of changes are committed in a map in a short period of time, the write-through caching can result in back-end overload, and the latter becomes a bottleneck.

Figure 5-9 illustrates a synchronous map entry insert with write-through enabled.

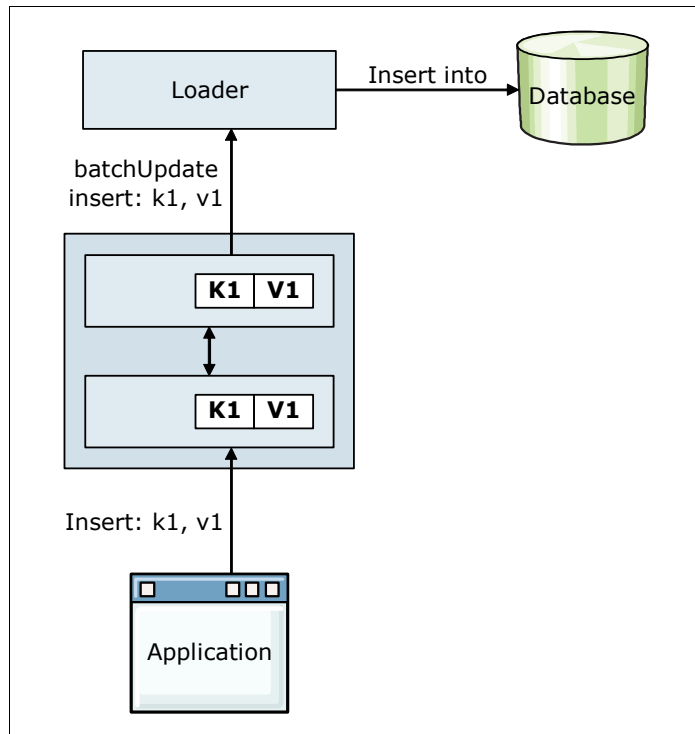


Figure 5-9 Synoptic of a synchronous map entry insert with write-through enabled

To avoid this problem, you can configure a loader plug-in to apply changes to the back-end system in an asynchronous way. With this mechanism, called *write-behind caching*, all changes are buffered in an ObjectGrid queue map. After either of these two criterion is reached, a time-based delay, or a maximum count of pending changes, the Loader `batchUpdate()` method is called on a separate thread to apply all the pending changes to the back-end system. Unlike a write-through scenario, the changes usually are from a number of committed transactions and thus are a larger “batch” of updates than for write-through.

Update timing note: The updates to the back-end system do not occur exactly when one of the criterion is reached. There can be small delay and variations with time or count. In all cases, only fully committed transactions will contribute their changes to what is written.

Figure 5-10 illustrates a synchronous map entry insert with write-behind enabled.

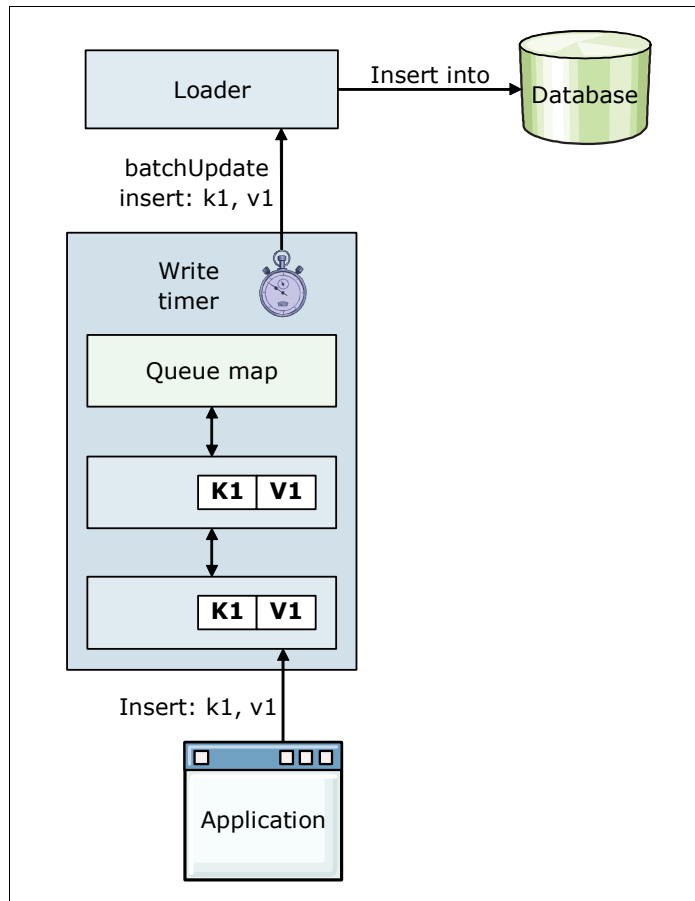


Figure 5-10 Synoptic of a synchronous map entry insert with write-behind enabled

To configure the write-behind feature of a loader plug-in, look in the ObjectGrid descriptor XML file and locate the backingMap element whose pluginCollectionRef refers to this Loader. Set the writeBehind attribute with a value with the format "[T(time)][;][C(count)]", with time and count being both positive integers.

Write-through caching and write-behind caching are mutually exclusive. If you do not specify the writeBehind attribute, the default write-through caching will be enabled. If you specify the writeBehind attribute with an empty string, the write-behind caching will be enabled with the default values of 300 seconds for the update time and a value of 1000 for the update count (the equivalent of specifying "T300;C1000"). Example 5-16 shows an example of a write-behind caching configuration.

Example 5-16 Write-behind caching example

```

<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="UserGrid" txTimeout="60">
      <bean id="TransactionCallback"
        className="com.ibm.websphere.objectgrid.jpa.JPATxCallback">
        <property name="persistenceUnitName" type="java.lang.String"
          value="userPUDB2DS"/>
      </bean>
    </objectGrid>
  </objectGrids>
</objectGridConfig>

```

```

        <backingMap name="User" pluginCollectionRef="User"
            writeBehind="T5;C900" lockStrategy="PESSIMISTIC"/>
    </objectGrid>
</objectGrids>
<backingMapPluginCollections>
    <backingMapPluginCollection id="User" >
        <bean id="Loader" className="com.ibm.websphere.objectgrid.jpa.JPALoader">
            <property name="entityClassName" type="java.lang.String"
                value="com.ibm.websphere.sample.xs.inlinebuffer.model.User"/>
        </bean>
    </backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>

```

Write-behind caching considerations

Configuring the write-behind caching is easy, but you must be aware of certain implications in the background. When using write-through caching, ObjectGrid transactions enclose the back-end transactions, and transaction events display in the following chronological order:

1. ObjectGrid transaction starts.
2. Back-end transaction starts.
3. Back-end transaction ends.
4. ObjectGrid transaction ends.

If an error occurs at any point, both transactions can be rolled back, thus leaving the data integrity intact.

With write-behind caching enabled, ObjectGrid transactions complete before back-end transactions start and are then decoupled from back-end transactions. The consequence of this action is that you must set up your own mechanism for dealing with permanent failures in the loader's ability to write to your backing store. A transitory failure, such as a temporary communication problem, is handled by the WebSphere eXtreme Scale transaction commit retry logic (see "Handling errors" on page 104).

Another consideration of which you must be aware is when separate maps are configured with write-behind caching enabled and the objects that they hold have a relationship between them. For example, suppose that you have Map1 and Map2 and that, in the same WebSphere eXtreme Scale transaction, key K1 in Map1 and key K2 in Map2 are updated. Everything goes fine, and a little bit later, the write-behind thread of each map is awakened and starts to write the pending updates in the back-end system. Because these updates occurred in separate transactions, it is possible that integrity constraints will be violated. This violation has an impact on the way that you have to design referential integrity constraints, which must be out-of-order tolerant. If you must have strict ordering, you have the option of designing a Loader, which you attach only to the BackingMap for the root object. This Loader's batchUpdate method can then use the root objects that it is given to obtain the related child objects (by acting as an WebSphere eXtreme Scale client to those other BackingMaps) and can then write the updated objects in whatever order is required by your back end's integrity constraints. If you have partitioned your data so that the root and all related child objects are in the same partition (which you probably had to do in order to commit them to the grid in a single transaction), your Loader code can access these child objects locally at an extremely high speed. Hint: Use `TxID.getSession().getMap(mapName)` to access the child maps.

Handling errors

The loader plug-in can fail to push updates because of a data failure (duplicate key or optimistic collision, for example) or because it cannot communicate with the back-end system. The loader implementation must handle these errors with care and must throw or re-throw the right exception. For a data failure, a `LoaderException` or an `OptimisticCollisionException` must be thrown. For a communication problem, the loader must throw a `LoaderNotAvailableException`.

WebSphere eXtreme Scale handles these two exceptions differently. On a `LoaderNotAvailableException`, the write-behind loader retries the update every 15 seconds. On a `LoaderException`, the write-behind loader will remove the update from the queue map and will retry the update by calling your `batchUpdate` method, one record (that is, one entry in the map) at a time.

If another `LoaderException` is thrown, an entry is created in the failed update map. A common mistake is to throw a `LoaderException` when a communication problem arises while a `LoaderNotAvailableException` needs to be thrown. The updates will become failed updates while they need to be retried, losing one of the key points of the write-behind plug-in: the back-end failure isolation and (when possible) recovery.

5.4.2 OptimisticCallback plug-ins

An `OptimisticCallback` is a plug-in that allows your custom logic to decide “who wins” when you configure `lockStrategy=OPTIMISTIC` and a conflict occurs. In optimistic locking strategy, it is possible for two transactions to attempt to commit changes to the same entry in a map. One of them commits first, then when the second one commits, the conflict is detected and someone (this plug-in, possibly written by you) must decide what to do. For more details about what happens in optimistic locking, see 6.3, “Locking performance preferred practices” on page 122. For further details about the how and why of `OptimisticCallback` plug-ins, see the `OptimisticCallback` plug-in section of the *WebSphere eXtreme Scale Programming Guide* at this website:

ftp://ftp.software.ibm.com/software/webserver/appserv/library/v71/xSprogguide_PDF.pdf

First, to enable optimistic locking on the entries in an WebSphere eXtreme Scale map, you choose (or add) a *version field* in the class for the *value* objects. WebSphere eXtreme Scale permits you to use a number of Java types as your version field. See the *WebSphere eXtreme Scale Programming Guide* for the various types that you can use, including `int`, `long`, `java.util.Date`, and several others. In our example, the gridded value object has a `Date` attribute called `xsVersion`, and we will use it as the version field.

The next step is to write an `OptimisticCallback` plug-in. Here is an example of an `OptimisticCallback` plug-in that works for our object.

Example 5-17 Example OptimisticCallback plug-in

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Date;
import com.ibm.websphere.objectgrid.plugins.OptimisticCallback;
/**
 * An example OptimisticCallback plug-in for a map containing entries whose key is
 * of type String (or whatever) and
 * whose value is of type ExamplePOJO. ExamplePOJO has an attribute called
```

```

* xsVersion of type java.util.Date
* (this example assumes that millisecond accuracy is sufficient).
*
*/
public class ExampleOptimisticCallbackImpl implements OptimisticCallback {

    /* (non-Javadoc)
    * @see
    com.ibm.websphere.objectgrid.plugins.OptimisticCallback#getVersionedObjectForValue
    (java.lang.Object)
    */
    public Object getVersionedObjectForValue(Object value) {
        if (value == null)
        {
            return null;
        }
        else
        {
            ExamplePOJO pojo = (ExamplePOJO) value;
            return pojo.getXsVersion();
        }
    }

    /* (non-Javadoc)
    * @see
    com.ibm.websphere.objectgrid.plugins.OptimisticCallback#updateVersionedObjectForVa
    lue(java.lang.Object)
    */
    public void updateVersionedObjectForValue(Object value) {
        if ( value != null )
        {
            ExamplePOJO pojo = (ExamplePOJO) value;
            pojo.setXsVersion(new Date());
        }
    }

    /* (non-Javadoc)
    * @see
    com.ibm.websphere.objectgrid.plugins.OptimisticCallback#inflateVersionedValue(java
    .io.ObjectInputStream)
    */
    public Object inflateVersionedValue(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
        long dateAsLong = in.readLong();
        Date date = new Date();
        date.setTime(dateAsLong);
        return date;
    }

    /* (non-Javadoc)
    * @see
    * com.ibm.websphere.objectgrid.plugins.OptimisticCallback#serializeVersionedValue
    * (java.lang.Object, java.io.ObjectOutputStream)
    */
    public void serializeVersionedValue(Object value, ObjectOutputStream out)

```

```

        throws IOException {
            Date date = (Date)value;
            out.writeLong(date.getTime());
        }
    }
}

```

The final step is to configure our `ExampleOptimisticCallbackImpl` class so that WebSphere eXtreme Scale knows to use it. Example 5-18 shows how we can change an `objectgrid.xml` to achieve this configuration (the parts in italics are the changes).

Example 5-18 ExampleOptimisticCallbackImpl class

```

...
<objectGrids>
  <objectGrid name="MyGrid">
    <backingMap name="MyBackingMap" readOnly="false" lockStrategy="OPTIMISTIC"
pluginCollectionRef="MyBackingMapPlugins"/>
  </objectGrid>
</objectGrids>
<backingMapPluginCollections>
  <backingMapPluginCollection id="MyBackingMapPlugins">
    <bean id="OptimisticCallback"
      className="com.mycorp.myapp.ExampleOptimisticCallbackImpl"/>
  </backingMapPluginCollection>
</backingMapPluginCollections>
...

```

5.4.3 MapEventListener plug-ins

A *MapEventListener* is a plug-in that allows you to execute custom logic when certain events happen. If you want to be notified whenever an entry is evicted (for whatever reason, such as if the time-to-live value that you configured has expired), you can write a *MapEventListener* implementation that will be called when an entry is evicted.

Example 5-19 shows an example of a simple *MapEventListener*, which logs to `SystemOut.log` when various methods are called.

Example 5-19 Example MapEventListener

```

import java.util.Date;
import com.ibm.websphere.objectgrid.plugins.MapEventListener;
public class ExampleMapEventListener implements MapEventListener {

    public void entryEvicted(Object key, Object value) {
        System.out.println("ExampleMapEventListener: At " + new Date() + " an entry
with key " + key + " and value " + value + " has been evicted.");
    }

    public void preloadCompleted(Throwable arg0) {
        // Not much interesting to do here
    }
}

```

To configure your grid so that this *MapEventListener* is used, you must modify your *ObjectGrid* descriptor XML file (the deployment XML file does not have to be modified) to resemble Example 5-20 on page 107 (the changes are in italics).

```
...
<objectGrids>
  <objectGrid name="MyGrid">
    <backingMap name="MyBackingMap" readOnly="false" lockStrategy="OPTIMISTIC"
pluginCollectionRef="MyBackingMapPlugins" />
  ...
</objectGrids>
<backingMapPluginCollections>
  <backingMapPluginCollection id="MyBackingMapPlugins">
    <bean id="MapEventListener"
className="com.mycorp.myapp.ExampleMapEventListener" />
  ...
</backingMapPluginCollections>
```

5.4.4 Indexing plug-ins

An *indexing plug-in* enables certain built-in behaviors of WebSphere eXtreme Scale. The behavior allows you to greatly improve the performance of certain queries (as described in 6.5.2, “Using indexes” on page 134). The queries that can be improved are any that have a “WHERE” clause containing one or a small set of object attributes. The idea is that you tell WebSphere eXtreme Scale (via XML) to construct and maintain an alternate index with those attributes.

For example, your map has a key of type AccountId and a value of type Account. Within Account is a String attribute called “name” with a value like “John Smith”. Most of the time, you get an Account using AccountId, but sometimes you want to get an account by name. To use an WebSphere eXtreme Scale index for this capability, you use Account.name as the attribute that you want to index. The drawback of this approach is that there can potentially be several Accounts with a name of “John Smith”.

To ensure that your query “WHERE a.name=’John Smith’ ” gets all the entries, WebSphere eXtreme Scale must maintain an index on every partition. Consider that each of those indexes can potentially have an entry for “John Smith” and there can be multiple account numbers in that partition that all have a name attribute of “John Smith”. Using the index is much faster than searching the whole partition, but WebSphere eXtreme Scale must still check every partition. The more partitions that we have, as we scale our grid up, the worse things get (not scalable).

You know by now that the fastest way to get data from a grid is by key, because it is a single “hop” from the client to the one container that holds the data. WebSphere eXtreme Scale indexes are easy to configure, but they make *N* hops. Can we somehow achieve our goals with less than *N* hops?

One possibility is to use the “reverse index” pattern. This pattern does require a small bit of code on our part but the resulting performance goes from *N* hops to at most 2 hops. No matter how many AccountId/Account entries we put in the grid, getting any Account by name will take no more than 2 hops - and we are linearly scalable again.

The reverse index pattern is simple in concept and not hard in implementation. It depends on you having a common bit of code (perhaps part of a persistence or DAO framework in your application) that you call to insert or update an Account. Typically, that code would insert an AccountId/Account pair to the grid. To add a reverse index, that code must now also insert a String/AccountId pair to a new map called, for example, AccountNameRIndexMap. Whenever you update the name attribute in an Account you must now also remove the old

String/AccountId pair from the AccountNameRIndexMap and insert a new pair. This is no problem for the common case where reads far outnumber inserts or updates.

The one drawback of the reverse index pattern, which may or may not be a problem for you, is that it may be difficult to commit both the change to the main map and the change to the Rindex map in the same transaction. Since we started with the assumption that gets by name are less frequent, it may be no problem that you must use one transaction to commit the main map change and a different transaction for the Rindex. Many customers have found the reverse index pattern an excellent technique to implement an alternate index while keeping linear scalability.



Performance planning for application developers

This chapter describes performance considerations for application developers and capacity planning for system administrators. In this chapter, we describe the preferred practices to improve the performance of an ObjectGrid map.

For our purposes, we implement these preferred practices only in the context of the application and its architecture. Every application and environment use a separate solution for performance. WebSphere eXtreme Scale provides built-in customizations to help improve performance, but you can also help improve performance within the application architecture.

This chapter includes the following topics:

- ▶ Copy mode method preferred practices
- ▶ Evictor performance preferred practices
- ▶ Locking performance preferred practices
- ▶ Serialization performance
- ▶ Query performance tuning

6.1 Copy mode method preferred practices

Figure 6-1 is an example of the flow of a request for data in a simple grid operation. Steps 1 through 11 take you through a typical flow. When an application receives an object from an ObjectMap, WebSphere eXtreme Scale performs a deep copy on the object value to ensure that the copy in the BackingMap map maintains data integrity. The application can then modify the object value safely. When the transaction commits (as shown in step 7 in Figure 6-1), the copy of the object value in the BackingMap is updated to the new modified value, and the application stops using the ObjectMap value from that point forward.

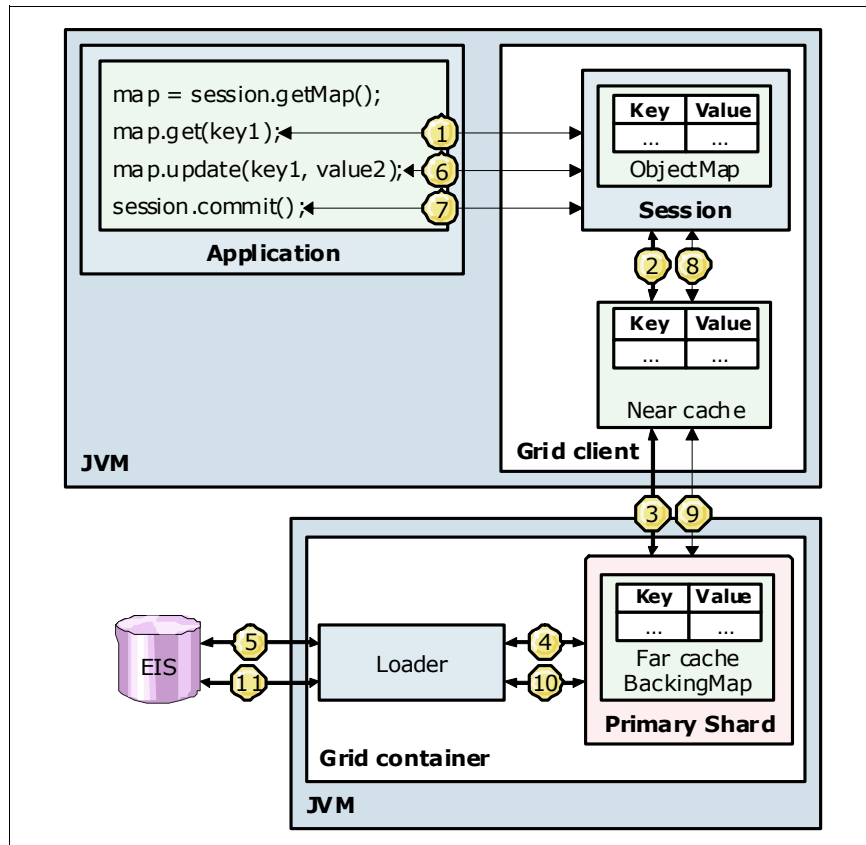


Figure 6-1 Request flow in a simple grid operation

In fact, there are several internal maps involved in accessing grid data:

- ▶ A transaction-scoped ObjectMap, containing only the entries accessed within the current transaction of a session.
- ▶ An optional near cache, containing all entries accessed by any transactions on this client Java virtual machine (JVM) since the JVM was started (minus any entries that have been evicted).
- ▶ The BackingMap, sometimes called the “*far cache*” to contrast it with the near cache. Logically, the BackingMap contains all entries for this map. The BackingMap consists of many primary shards (and perhaps replica shards). Any particular entry is in one and only one primary shard (not counting multi-master replication).
- ▶ Your enterprise information system (EIS) back end, which is not really a map but contains your underlying data. If you have a loader, it interfaces between the EIS and the BackingMap, as described elsewhere.

Your choice of copy mode affects when WebSphere eXtreme Scale performs the deep copies between the BackingMap and near cache and between the near cache and transaction-scoped cache.

You can have WebSphere eXtreme Scale copy the object again at the commit phase to make a private copy; however, in this case, the performance benefit of this action is traded against requiring that the application programmer not use the value after the transaction commits. Determining which copy mode setting works best for your deployment requirements affects the overall performance. You can tune the number of copies by defining the CopyMode attribute of the BackingMap or ObjectMap object.

The CopyMode attribute has the following values:

- ▶ COPY_ON_READ_AND_COMMIT
- ▶ COPY_ON_READ
- ▶ COPY_ON_WRITE
- ▶ NO_COPY
- ▶ COPY_TO_BYTES

You can set the CopyMode attribute with the ObjectGrid XML file for a map, as shown in the following example:

```
<backingMap name="myMap" copyMode="NO_COPY" />
```

Here is a summary of how copyMode affects the behavior of WebSphere eXtreme Scale. The subsequent sections cover each copyMode choice in more detail.

CopyMode: If copyMode is COPY_TO_BYTES, it behaves like COPY_ON_READ_AND_COMMIT, except for also skipping container-side serialization and inflation, as stated in the following information.

The following steps show how copyMode affects the behavior of WebSphere eXtreme Scale:

1. A client calls get.
2. The request goes to the container:
 - For a local container or for a near cache, the following actions occur:
 - If COPY_ON_READ_AND_COMMIT or COPY_ON_READ, the requested object is copied. The client receives the copy.
 - If NO_COPY, the client receives a reference to the value object in the local container or near cache.
 - If COPY_ON_WRITE, the requested object has a proxy made for it. The client receives the proxy object, which in turn refers to the object in the local container or near cache.
 - For a remote container, the requested object is serialized (skipped if copyMode is COPY_TO_BYTES) and sent over the wire to the client. The client receives the serialized response and inflates the value. Because we have just inflated the value, it is already a copy from the wire. Therefore, we do not make another copy.
3. A client calls insert, update, remove, and so forth. There is no communication to any remote container.

For a local container or for a near cache, the written object is copied if COPY_ON_WRITE. The proxy that is held by the client no longer refers to the object in the local container or near cache.

4. A client commits the transaction:

- For a local container or for a near cache, the committed values (and keys in the case of inserts) are copied if `COPY_ON_READ_AND_COMMIT`. The reference that the client still holds no longer refers to the object in the local container or near cache.

If other than `COPY_ON_READ_AND_COMMIT`, the reference that the client still holds does still refer to the object in the local container or near cache, but the client must not make use of it anymore.

- For a remote container, the committed values (and keys in the case of inserts) are serialized and go over to the container, and the key-value pairs are inflated (skipped if `COPY_TO_BYTES`) on the server. The value that we received is already a copy from the wire. Therefore, we do not make another copy. However, for a brief time (after a value has been received and before the primary partition has been updated with the new value), the data from the committing transaction and the prior data co-exist in memory on the container JVM.

If you use agents, the agents execute locally to the container of the objects on which they will act, and the behavior is the same as for a client and a local container, as described previously.

WebSphere eXtreme Scale V6.1.0.5 and beyond behave in this manner. Prior to V6.1.0.5, copies were made far more often in the case of remote containers.

6.1.1 COPY_ON_READ_AND_COMMIT mode

`COPY_ON_READ_AND_COMMIT` mode is the default mode. This mode ensures that an application does not contain a reference to the value object that is in the `BackingMap` (of course, this situation is only possible for embedded grids where the `BackingMap` is in the same JVM as the client application. For distributed grids, as pictured in Figure 6-1 on page 110, the client never works with the value object actually in the `BackingMap`. The client can, however, work with the value in the near cache, which has similar consequences). Instead, the application always works with a copy of the value that is in the `BackingMap`. `COPY_ON_READ_AND_COMMIT` mode ensures that the application can never inadvertently corrupt the data that is cached in the `BackingMap`.

When an application transaction calls an `ObjectMap.get` method for a given key and it is the first access of the `ObjectMap` entry for that key, a copy of the value is returned. When the transaction is committed, any changes committed by the application are copied as a new instance to the `BackingMap`. This approach ensures that the application does not have a reference to the committed value in the `BackingMap`. The object to which the application still has a reference is no longer known to WebSphere eXtreme Scale, and changes to that object do not affect the grid.

6.1.2 COPY_ON_READ mode

`COPY_ON_READ` mode improves performance over `COPY_ON_READ_AND_COMMIT` mode (for a local container or a near cache) by eliminating the copy that occurs when a transaction is committed. To preserve the integrity of the near cache or `BackingMap` data, the application must ensure that every reference that it has for an entry is destroyed after the transaction is committed. With this mode, the `ObjectMap.get` method returns a copy of the value instead of a reference to the value to ensure that changes made by the application to the value do not affect the near cache or `BackingMap` value until the transaction is committed. However, when the transaction does commit, a copy of the changes is not made. Instead, the reference to the copy that was returned by the `ObjectMap.get` method is stored in the near cache or `BackingMap`.

The application must destroy all map entry references after the transaction is committed. If the application fails to destroy the entry references, the application might cause the data that is cached in the BackingMap to become corrupted. If an application uses this mode and has problems, switch to `COPY_ON_READ_AND_COMMIT` mode to see if the problem still exists. If the problem no longer exists, the application is failing to destroy all of its references after the transaction has committed. Note that if you have a distributed grid and if you have disabled near cache, `COPY_ON_READ_AND_COMMIT` effectively behaves the same as `COPY_ON_READ`.

6.1.3 `COPY_ON_WRITE` mode

`COPY_ON_WRITE` mode improves performance over `COPY_ON_READ_AND_COMMIT` mode (for a local container or near cache) by eliminating the copy that occurs when the `ObjectMap.get` method is called for the first time by a transaction for a given key. The `ObjectMap.get` method returns a proxy to the value instead of a direct reference to the value object. The proxy ensures that a copy of the value is not made unless the application calls a set method on the value interface that is specified by the `valueInterfaceClass` argument. For `COPY_ON_WRITE` to work, your value object must implement a Java interface and your client code must refer only to the interface type; WebSphere eXtreme Scale uses your interface for its proxy. The value objects must also follow the JavaBeans pattern for setter methods.

The proxy provides a copy on write implementation. The application must still call `update(key,value)` if it updates a value. However, when a transaction commits, the BackingMap examines the proxy to determine whether a copy was made as a result of a set method being called. If a copy was made, the reference to that copy is stored in the BackingMap. The advantage of this mode is that a value is never copied on a read or at commit if the transaction does not call a set method to change the value.

`COPY_ON_WRITE` also permits an application to examine a value prior to commit to see what attributes were modified, if any. The proxy that the application holds implements not only your own interface but also WebSphere eXtreme Scale's `ValueProxyInfo`. A method of the latter returns a list (by name) of the value's attributes that were modified.

6.1.4 `NO_COPY` mode

`NO_COPY` mode allows an application to declare that it never modifies a value object that is obtained using an `ObjectMap.get` method in exchange for performance improvements. If this mode is used, no copy of the value is ever made. If the application modifies values, the data in the BackingMap is corrupted.

`NO_COPY` mode is useful for read-only maps where data is never modified by the application. This mode is not typically recommended to application developers, because it allows direct access to editing the grid data, regardless of transactions and locks. It is only safe to use this mode with read-only data, or where only one JVM at a time, and only one thread on that JVM at a time, will change a given value. An example of this mode is HTTP session management.

6.1.5 `COPY_TO_BYTES` mode

You can store the key value pairs in a BackingMap in a byte array instead of plain old Java object (POJO) form, which reduces the memory footprint that a large graph of objects can consume. By reducing a complicated graph of objects to a byte array, only one object is maintained in the heap instead of several objects. With this reduction of the number of objects in the heap, the Java run time has fewer objects to search for during garbage collection. When using byte arrays, note that having an optimized serialization mechanism is critical to seeing a reduction of memory consumption. `COPY_TO_BYTES` can also save significant

processing time, because half of the serialize and inflate code is no longer executed for get or commit. For all but the simplest objects, the performance improvement can be substantial.

The default copy mechanism that is used by WebSphere eXtreme Scale is serialization, which is expensive, for example, if using the default copy mode of `COPY_ON_READ_AND_COMMIT`, a copy is made both at read time and at commit time. Instead of making a copy at read time, with byte arrays, the value is inflated from bytes. In addition, instead of making a copy at commit time, the value is serialized to bytes. Using byte arrays results in equivalent data consistency to the default setting with an increase in speed and a possible reduction of the memory that is used.

You can enable byte array maps with the ObjectGrid XML file by modifying the `CopyMode` attribute that is used by a map to the setting `COPY_TO_BYTES`, which is shown in the following example:

```
<backingMap name="byteMap" copyMode="COPY_TO_BYTES" />
```

You must consider whether to use byte array maps in a given scenario. Although you can improve performance and reduce memory use, other aspects of processor use can increase when you use byte arrays. Consider the following factors before choosing to use the byte array map function:

- Object type

Comparatively, serialization speed and memory reduction might not be possible when using byte array maps for certain object types. Consequently, several types of objects exist for which there is rarely a benefit from the use of byte array maps. If you are using any of the Java primitive wrappers as values or a POJO that does not contain references to other objects (only storing primitive fields), the number of Java objects is already as low as possible (that is, there is only one). With regard to serialization performance, the fewer references to other objects you have, the faster serialization and inflation will be and the smaller the improvement from skipping half the processing. As for most things, run performance tests to determine the benefit you will gain.

Byte array maps are more beneficial for object types that contain other objects or collections of objects where the total number of POJOs is greater than one. For example, if you have a customer object that has a business address and a home address and a collection of orders, you can reduce the number of objects in the heap and the number of bytes that are used by those objects by using byte array maps. These objects also take non-trivial time to serialize and inflate, and the huge number of objects accessed in a typical grid magnifies any small improvement. See 6.4, “Serialization performance” on page 127 for more details.

- Local access

When using other copy modes, WebSphere eXtreme Scale can optimize when copies are made if objects can be cloned with the default `ObjectTransformer` or when a custom `ObjectTransformer` is provided with an optimized `copyValue` method. Compared to the other copy modes, copying on reads, writes, or commit operations has additional costs when accessing objects locally. For example, if you directly access a local or server ObjectGrid instance, the access and commit time increases when using byte array maps, because the byte array must be inflated back into a Java object. For this reason, applications that make significant use of agents might not see a net benefit from `COPY_TO_BYTES`.

- Plug-in interactions

With byte array maps, objects are not inflated when communicating from a client to a server unless the server needs the POJO form. Plug-ins that interact with the map value experience a reduction in performance due to the requirement to inflate the value.

- Indexes and queries

When objects are stored in POJO format, the cost of performing indexing and querying is minimal, because the object does not need to be inflated before the index or query engine can act on it. When using a byte array map, you have the additional cost of inflating the object. In general, if your application uses indexes or queries often, do not use byte array maps unless you run queries only on key attributes.

- Optimistic locking

When using the optimistic locking strategy, you have an additional cost during updates and invalidate operations. This cost results from having to inflate the value on the server to get the version value to do optimistic collision checking. If you use optimistic locking only to guarantee fetch operations and you do not need optimistic collision checking, you can use the `com.ibm.websphere.objectgrid.plugins.builtins.NoVersioningOptimisticCallback` class to disable version checking. However, this approach might be slow, and it requires your object to implement `equals()`.

- Loader

With a loader, you also have the cost in the WebSphere eXtreme Scale run time from inflating and reserializing the value when it is used by the loader. You can still use byte array maps with loaders, but consider the cost of making changes to the value in such a scenario. For example, you can use the byte array feature in the context of a read mostly cache. In this case, the benefit of `COPY_TO_BYTES` in performance and memory outweighs the cost that is incurred from using byte arrays on insert and update operations.

- ObjectGridEventListener

When using the `transactionEnd` method in the `ObjectGridEventListener` plug-in, you have an additional cost on the server side for remote requests when accessing a `LogElement`'s `CacheEntry` or current value. If the implementation of the method does not access these fields, you do not have the additional cost.

As you have seen, the common theme for all these considerations is how often your byte arrays will have to be inflated back into objects for use by agents, queries, and so forth versus how often you will save processing time by skipping inflation when it is not really necessary.

6.2 Evictor performance preferred practices

WebSphere eXtreme Scale provides a default mechanism for evicting cache entries and a plug-in for creating custom evictors to control the growth of cache grid size. An *evictor* controls the membership of entries in each `BackingMap` instance.

We can distinguish three kinds of evictors, depending on the algorithm that they use:

- Evictors based on a lifetime
- Evictors based on a number of entries
- Evictors based on the percent of your maximum heap (`Xmx`) currently in use

6.2.1 Default (time-to-live) evictor

A default evictor is created with every backing map that removes entries based on a time-to-live (TTL) concept. The TTL evictor is not active by default but can be enabled programmatically using the BackingMap interface, or it can be enabled through the ObjectGrid descriptor XML file.

When enabled, the TTL evictor uses an algorithm based on an expiration delay (time-to-live), which is added to a reference time. The reference time can be the creation time of an entry in the map or the last time that an entry was accessed or updated. This behavior is defined by the TTLtype attribute (programmatically) or the ttlEvictorType attribute in the XML file. The TTL type is one of the following types:

- ▶ **NONE**
Specifies that entries never expire and, therefore, are never removed from the map, effectively disabling the evictor
- ▶ **CREATION_TIME**
Specifies that entries are evicted, depending upon when they were created
- ▶ **LAST_ACCESS_TIME**
Specifies that entries are evicted, depending upon when they were last accessed and whether they were read or updated at the time
- ▶ **LAST_UPDATE_TIME**
Specifies that entries are evicted, depending upon when they were last updated

To enable the TTL evictor for a specific map, set the TTL type and a time-to-live value in seconds. This value is the expiration delay, which will be used to determine when to evict entries.

Example 6-1 shows an example of enabling TTL evictors for each of three maps using the ObjectGrid descriptor XML file.

Example 6-1 Enabling TTL evictors

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
  <objectGrid name="grid1">
    <backingMap name="map1" ttlEvictorType="NONE" />
    <backingMap name="map2" ttlEvictorType="LAST_ACCESS_TIME"
      timeToLive="1800" />
    <backingMap name="map3" ttlEvictorType="CREATION_TIME"
      timeToLive="1200" />
  </objectGrid>
</objectGrids>
```

If you use CREATION_TIME, an entry is evicted when its time from creation equals its timeToLive value. If you set the timeToLive value to 10 seconds, the entry is evicted automatically 10 seconds after it was inserted. Take caution when setting this value. This evictor type is best used when reasonably high numbers of additions to the cache exist that are used only for a set amount of time. With this strategy, anything that is created is removed after the set amount of time. CREATION_TIME is useful in scenarios, such as refreshing

stock quotes every 20 minutes or less. Suppose a web application obtains stock quotes, and getting the most recent quotes is not critical. In this case, the stock quotes are cached in an ObjectGrid for 20 minutes. After 20 minutes, the ObjectGrid map entries expire and are evicted. When a client tries to access an evicted entry, the ObjectGrid map uses the loader plug-in to refresh the map data with fresh data from the database. The value of 20 minutes might have been chosen, for example, because the database is updated every 20 minutes with the most recent stock quotes.

If you use `LAST_ACCESS_TIME` or `LAST_UPDATE_TIME`, set `timeToLive` to a lower number than if you use `CREATION_TIME`. The `timeToLive` counter is reset every time that it is accessed. If `timeToLive` is equal to 15 and if an entry has existed for 14 seconds but then gets accessed, it does not expire again for another 15 seconds. If you set `timeToLive` to a relatively high number, many entries might never be evicted. However, if you set the value to a lower number, such as 15 seconds, entries might be removed when they are not accessed often. The `LAST_ACCESS_TIME` or `LAST_UPDATE_TIME` type is useful in scenarios, such as holding session data from a client that uses an ObjectGrid map. Session data must be destroyed if the client does not use the session data for a certain period of time. For example, the HTTP (web) servers might be configured so that user sessions time out after 30 minutes of no activity by the client. In this case, using `LAST_ACCESS_TIME` or `LAST_UPDATE_TIME` with `timeToLive` set to 30 minutes is appropriate for this application.

6.2.2 Pluggable evictors with LFU and LRU properties

The default TTL evictor uses an eviction policy that is based on time, and the number of entries in the BackingMap has no affect on the expiration time of an entry. You can use an optional pluggable evictor to evict entries based on the number of entries that exist instead of based on time. Remember that if your entries are of a variable size (which most entries are), evicting based on the number of entries might not help prevent out-of-memory errors; you might prefer memory-based eviction (see 6.2.3, “Memory-based eviction” on page 120).

There are two algorithms that are available to define which entries must be evicted:

- ▶ Least recently used (LRU)
- ▶ Least frequently used (LFU)

An *LFU evictor* removes entries from the map that are used infrequently. The entries of the map are spread over a set amount of binary heaps. As the usage of a particular cache entry grows, it becomes ordered higher in the heap. When the evictor attempts a set of evictions, it removes only the cache entries that are located lower than a specific point on the binary heap. As a result, the least frequently used entries are evicted.

An *LRU evictor* follows the same concepts as the LFU evictor with a few differences. The LRU uses a first in, first out (FIFO) queue instead of a set of binary heaps. Every time that a cache entry is accessed, it moves to the head of the queue. Consequently, the front of the queue contains the most recently used map entries and the end of the queue becomes the least recently used map entries. For example, the A cache entry is used 50 times, and the B cache entry is used only once immediately after the A cache entry. In this situation, the B cache entry is at the front of the queue because it was used most recently, and the A cache entry is at the end of the queue. The LRU evictor evicts the cache entries that are at the end of the queue, which are the least recently used map entries. For both LFU and LRU evictors, the exact number of entries evicted is variable. Each partition is told to evict a certain percentage of its entries, and it does so independently of any other partition. The net result, however, is that your grid is back under the maximum number of entries that you specified.

To enable a pluggable evictor for a map, complete the following tasks in your ObjectGrid descriptor XML file:

- ▶ Add the `pluginCollectionRef` attribute in the `backingMap` element for the map for which you want to enable the evictor. The value of the attribute must be the ID of the collection of plug-ins for the map.
- ▶ Add a new bean element with the Evictor ID as a child of the `backingMapPluginCollection` element. In the bean element, set the attribute `className` with the value of the built-in plug-in that you want to use:
 - `com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor`
 - `com.ibm.websphere.objectgrid.plugins.builtins.LFUEvictor`
- ▶ Add three property elements as children of the bean element to customize the evictor. Each property element must have at least a name, a type, and a value attribute:
 - For both plug-ins, add a `maxSize` and `sleepTime` property.
 - For an LRU plug-in, add the `numberOfLRUQueues` property. For the LFU plug-in, add the `numberOfHeaps` property.

All properties are of type `int`.

Tip: Because there is no specification for default values, it is a good idea to define all the properties of a plug-in.

Example 6-2 provides an example of the definition of the LRU and LFU evictors.

Example 6-2 Definition of the LRU and LFU evictors

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="grid1">
      <backingMap name="map1" pluginCollectionRef="LRU" />
      <backingMap name="map2" pluginCollectionRef="LFU" />
    </objectGrid>
    <backingMapPluginCollections>
      <backingMapPluginCollection id="LRU">
        <bean id="Evictor"
          className="com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor">
          <property name="maxSize" type="int" value="1000" description="set max
size for each LRU queue" />
          <property name="sleepTime" type="int" value="15" description="evictor
thread sleep time" />
          <property name="numberOfLRUQueues" type="int" value="53"
description="set number of LRU queues" />
        </bean>
      </backingMapPluginCollection>
      <backingMapPluginCollection id="LFU">
        <bean id="Evictor"
          className="com.ibm.websphere.objectgrid.plugins.builtins.LFUEvictor">
          <property name="maxSize" type="int" value="2000" description="set max
size for each LFU heap" />
          <property name="sleepTime" type="int" value="15" description="evictor
thread sleep time" />
        </bean>
      </backingMapPluginCollection>
    </backingMapPluginCollections>
  </objectGrids>
</objectGridConfig>
```

```

        <property name="numberOfHeaps" type="int" value="211" description="set
number of LFU heaps" />
    </bean>
</backingMapPluginCollection>
</backingMapPluginCollections>
</objectGrids>

```

Understand the following properties:

► Number of heaps

When using the LFU evictor, all of the cache entries for a particular map are ordered over the number of heaps that you specify, improving performance and preventing the evictions from synchronizing on one binary heap that contains all of the ordering for the map. Having more heaps also speeds up the time that is required for reordering the heaps, because each heap has fewer entries. Set the number of heaps to 10% of the number of entries in your BaseMap.

► Number of queues

When using the LRU evictor, all of the cache entries for a particular map are ordered over the number of LRU queues that you specify, improving performance and preventing the evictions from synchronizing on one queue that contains all of the ordering for the map. Set the number of queues to 10% of the number of entries in your BaseMap.

► MaxSize property

When an LFU or LRU evictor begins evicting entries, it uses the MaxSize evictor property to determine how many binary heaps or LRU queues to evict. A simple way to think of this property is that the maximum number of objects that you want in your map is *roughly* maxSize times the numberOfHeaps (or maxSize times the numberOfLRUQueues). For example, assume that you set the number of heaps or queues to have about 10 map entries in each map queue. If your MaxSize property is set to 7, the evictor evicts 3 entries from each heap or queue object to bring the size of each heap or queue back down to 7. The evictor only evicts map entries from a heap or queue when that heap or queue has more than the MaxSize property value of elements in it. Set the MaxSize to 70% of your heap or queue size. For this example, the value is set to 7. You can get an approximate size of each heap or queue by dividing the number of BaseMap entries by the number of heaps or queues used.

► SleepTime property

An evictor does not constantly remove entries from your map. Instead, it is idle for a set amount of time, checking the map only every n number of seconds, where n refers to the SleepTime property. This property also positively affects performance. Running an eviction sweep too often lowers performance because of the resources that are needed for processing them. However, not using the evictor often enough can result in a map that has entries that are not needed. A map full of entries that are not needed can negatively affect both the memory requirements and processing resources that are required for your map.

Setting the eviction sweep interval to 15 seconds is a preferred practice for most maps. If the map is written to frequently and is used at a high transaction rate, consider setting the value to a shorter time or lower value. If the map is accessed infrequently, you can set the time to a higher value.

Example 6-3 on page 120 shows the optional pluggable evictors (set programmatically) that provide commonly used algorithms for deciding which entries to evict when a BackingMap grows beyond a certain size limit.

Example 6-3 Using the LFU evictor

```
//Use ObjectGridManager to create/get the ObjectGrid. Refer to
// the ObjectGridManger section
ObjectGrid objGrid = ObjectGridManager.createObjectGrid("Grid");
BackingMap bMap = objGrid.defineMap("User");

//Set properties assuming 50,000 map entries
LFUEvictor someEvictor = new LFUEvictor();
someEvictor.setNumberOfHeaps(5000); // 10% of entries (50,000)
someEvictor.setMaxSize(7); // 7 * 5000 is 70% entries (50,000)
someEvictor.setSleepTime(15);
bMap.setEvictor(someEvictor);
```

Using the LRU evictor is similar to using an LFU evictor, as shown in Example 6-4.

Example 6-4 Using the LRU evictor

```
ObjectGrid objGrid = new ObjectGrid;
BackingMap bMap = objGrid.defineMap("User");

//Set properties assuming 50,000 map entries
LRUEvictor someEvictor = new LRUEvictor();
someEvictor.setNumberOfLRUQueues(5000); // 10% of entries (50,000)
someEvictor.setMaxSize(7); // 7 * 5000 is 70% of entries (50,000)
someEvictor.setSleepTime(15);
bMap.setEvictor(someEvictor);
```

6.2.3 Memory-based eviction

Memory-based eviction is supported only on Java Platform, Enterprise Edition V5 or later.

All built-in evictor plug-ins support a trigger based on a memory threshold. These plug-ins can be invoked automatically when the Java memory heap utilization reaches a certain value. This threshold can be used, for example, to avoid an `OutOfMemory` error that can occur before the other eviction conditions of the plug-in arise. To enable this feature, you must set the `evictionTriggers` attribute to `MEMORY_USAGE_THRESHOLD` in the `backingMap` element.

With this attribute set, you next need to decide what entries must be evicted when memory runs low. You can use the built-in TTL evictor; when memory is low, the `timeToLive` that you set will be temporarily reduced a few seconds at a time and entries older than that will be evicted until memory is acceptable again. Alternatively, you can also use the `LRUEvictor` or `LFUEvictor` plug-in; when memory is low, the evictor will be asked to evict entries until memory is alright again. You do not have to set any properties on the `LRUEvictor`/`LFUEvictor`; the mere presence of the evictor in your plug-in reference list is enough. If you set properties on the evictor that relate to a maximum number of entries, it acts both as a number-of-entries evictor and a memory-usage evictor.

When memory-based eviction is enabled on a `BackingMap` and when the `BackingMap` has any built-in evictor, the usage threshold is set to the default percentage of total memory (70%) if the threshold has not been set previously by you.

The memory-based eviction algorithm that is used by WebSphere eXtreme Scale is sensitive to the behavior of the garbage collection algorithm in use. The best algorithm for memory-based eviction is the IBM default throughput collector ("opt thruput"). Generation garbage collection algorithms might be good for other reasons, but you cannot use

memory-based eviction with them, because WebSphere eXtreme Scale depends on a JVM's MBean feature to actually monitor memory use and notify WebSphere eXtreme Scale when the threshold is reached.

To change the usage threshold percentage, set the `memoryThresholdPercentage` property on the container and server property files for WebSphere eXtreme Scale server processes. The default value is 70%.

During run time, if the memory usage exceeds the target usage threshold, any evictors will start evicting entries and try to keep memory usage beneath the target usage threshold. However, no guarantee exists that the eviction speed is fast enough to avoid a potential out-of-memory error if the system run time continues to consume memory quickly.

Example 6-5 provides an example of memory threshold utilization.

Example 6-5 Memory threshold utilization

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
  <objectGrid name="grid1">
    <backingMap name="map2" ttlEvictorType="CREATION_TIME" timeToLive="1800"
      evictionTriggers="MEMORY_USAGE_THRESHOLD"/>
  </objectGrid>
</objectGrids>
```

When a map is configured with the `evictionTriggers` attribute set to `MEMORY_USAGE_THRESHOLD`, the messages that are shown in Example 6-6 display in the standard JVM log file (`SystemOut.log`).

Example 6-6 JVM output messages when memory threshold is enabled

```
CWOB0J0046I: The memoryThresholdPercentage property was not provided in a server
properties file and the MemoryMXPoool bean was not set (current value = 0). A
default value of 70 will be used.
CWOB0J0034I: Memory utilization threshold percentage is set to 70 %.
CWOB0J0036W: Changing memory utilization threshold from 0K to 367001K for Java
heap memory pool.
CWOB0J0037W: Changing memory collection utilization threshold from 0K to 367001K
for Java heap memory pool.
```

As shown, the default memory threshold is 70% of the Java heap size.

To change the default to another value, you must provide a server properties file in which the property `memoryThresholdPercentage` is set. To understand how to set a server properties file, see this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extrmescale.admin.doc/rxscontprops.html>

Example 6-7 on page 122 shows the output messages when the memory threshold default value has been changed.

Example 6-7 JVM output messages when the memory threshold default value is changed

```
CWOBJS0046I: The memoryThresholdPercentage property was not provided in a server
properties file and the MemoryMXPool bean was not set (current value = 0). A
default value of 70 will be used.
CWOBJS0034I: Memory utilization threshold percentage is set to 70 %.
CWOBJS0036W: Changing memory utilization threshold from 0K to 367001K for Java
heap memory pool.
CWOBJS0037W: Changing memory collection utilization threshold from 0K to 367001K
for Java heap memory pool.
CWOBJS0034I: Memory utilization threshold percentage is set to 60 %.
CWOBJS0036W: Changing memory utilization threshold from 367001K to 314572K for
Java heap memory pool.
CWOBJS0037W: Changing memory collection utilization threshold from 367001K to
314572K for Java heap memory pool.
CWOBJS0034I: Memory utilization threshold percentage is set to 60 %.
```

Messages: The first message showing that the memoryThresholdPercentage was not provided is quite confusing. It is obviously contradicted with the next message showing that the memory threshold percentage was set to another value than the default. These messages are normal in the current product.

6.3 Locking performance preferred practices

An ObjectGrid BackingMap supports several locking strategies to maintain cache entry consistency. You can configure each BackingMap to use one of the following locking strategies:

- ▶ Pessimistic locking mode
- ▶ Optimistic locking mode
- ▶ None

Set the locking strategy on the BackingMap. You cannot change the locking strategy for each transaction. The following example shows how to set the locking mode on a map using the XML file:

```
<backingMap name="myMap" lockStrategy="PESSIMISTIC" />
```

6.3.1 Pessimistic locking strategy

The pessimistic locking strategy acquires locks for cache entries. Use this strategy when data is changed frequently. Any time that a cache entry is read, a lock is acquired and held conditionally until the transaction completes. The duration of certain locks can be tuned using transaction isolation levels for the session.

6.3.2 Optimistic locking strategy

Optimistic locking is the default configuration. This strategy improves both performance and scalability when compared to the pessimistic strategy. Use this strategy when your applications can tolerate occasional optimistic update failures. They perform better than they will by using the pessimistic strategy. This strategy is excellent for read operations and infrequent update applications.

Locks are held only for a short duration while data is being read from the cache and copied to the transaction. When the transaction cache is synchronized with the main cache, any cache objects that have been updated are checked against the original version. If the check fails, the transaction is rolled back, and an `OptimisticCollisionException` results.

6.3.3 None locking strategy

Use the none locking strategy for applications that are read only. The none locking strategy does not obtain any locks. Therefore, it offers the most concurrency, performance, and scalability. The none lock strategy is ideal for look-up tables or read-only maps.

6.3.4 Lock semantics for `lockStrategy=PESSIMISTIC, OPTIMISTIC, and NONE`

This material is based on the “Locking Strategy” and “Handling locks” sections of the *WebSphere eXtreme Scale V7.1 Programming Guide*. It applies to WebSphere eXtreme Scale V7.0 and 6.1, as well. Readers with knowledge of database locking semantics will find this material familiar.

An underlying principle that applies to all three strategies is that there are three lock modes, and each mode, when already held, allows or refuses requests for the other types in a certain way. The three lock modes are a property of an individual object that is stored in a `BackingMap` and, to an extent, each transaction. Any transaction at any point in time might not hold a lock of a given mode on a given object. Each object in the map is separately locked in one of the three ways (or no way) by a given transaction at any moment in time.

We describe the three lock modes and their behavior:

- ▶ S lock (shared lock): If a transaction holds an S lock on an object, other transactions can obtain an S lock or U lock on that object, but not an X lock on the object.
- ▶ U lock (upgradable lock): If a transaction holds a U lock on an object, other transactions can obtain an S lock on that object, but not a U lock or an X lock.
- ▶ X lock (exclusive lock): If a transaction holds an X lock on an object, other transactions cannot obtain an S lock, U lock, or X lock on that object.

There is also, of course, the state of no lock at all being held by a transaction for an object.

Another way to express these behaviors, somewhat more mathematically, is that the state of locks on a given object can only be:

(0-to-N S locks AND 0-to-1 U lock) OR 0-to-1 X lock

The locking behavior described here assumes that your transaction isolation setting is the default of repeatable read. You can change your transaction isolation to read committed or read uncommitted, which will modify the locking behavior of each lockingStrategy choice.

Another interesting thing to note is that you can lock an object, which is not (yet or ever) in the grid. You only need a valid key. If you call `myMap.getForUpdate(someKey)` and no such object exists (it returns null), you will then hold whatever lock you might have held if there had been an entry for that key. That lock will be freed in the same manner as if there had been an entry with that key. If another transaction tries to access that map with that key, you will have the same sort of lock conflict (or lack of conflict) as if there had been an entry. If the other transaction inserts an object into the map with that key, it will succeed or fail based on lock conflict rules in the same way. This function can be useful if you need to use locks to ensure only one transaction (that is, one thread) out of many transactions actually goes off to a

database to fetch data and inserts the object into the grid (this is commonly called “*avoiding a stampede*”).

Notice that this function also means that WebSphere eXtreme Scale can be used as a way to have cross-JVM mutually-exclusive locks (MUTEXs). These locks exist in basic Java for a single JVM, but not for use across JVMs.

PESSIMISTIC

The semantics for `lockStrategy=PESSIMISTIC` follow. Note that there is never a near cache for PESSIMISTIC.

1. The client gets the data from the container server. There are two scenarios:
 - Scenario 1: Client application (transaction T1) calls `get(key)` or `getAll()`:
 - i. Transaction (T1) goes to the container server for this object or objects and gets an S lock on the object or objects. Note that if another transaction T2 is in the middle of a commit, T2 will hold an X lock on the object and thus the S lock request will fail.
 - ii. T1 will block for *M* seconds waiting for the lock; *M* is configurable.
 - iii. If T1 is unable to get the S lock, an exception is thrown to the client. If T1 gets the S lock, the object or objects are copied to the T1 transaction scoped cache (“T1 cache” for short) on the client.
 - iv. The S lock is held until T1 commits. The `get` method call returns to the client with the S lock still held.
 - Scenario 2: Alternatively, client application (transaction T1) calls `getForUpdate(key)` or `getAllForUpdate()`:
 - i. Transaction (T1) goes to the container server for this object or objects and gets a U lock on the object or objects. Note that if another transaction T2 is in the middle of a commit, T2 will hold an X lock on the object and thus the U lock request will fail.
 - ii. T1 will block for *M* seconds waiting for the lock; *M* is configurable.
 - iii. If T1 unable to get the U lock, an exception is thrown to the client. If another transaction T2 holds a U lock, the U lock request for T1 will fail; the same block/wait and the same possible exception throw happens as for X lock.
 - iv. If T1 gets the U lock, the object or objects are copied to the T1 transaction scoped cache (“T1 cache”) on the client.
 - v. The U lock is held until T1 commits. The `get..` method call returns to the client with the U lock still held.
2. Client (T1) reads or modifies the object data. Any changes are made only to the object in the T1 cache. No locks are obtained or checked.
3. Client (T1) does `put(key)`, `putAll()`, `remove(key)`, `removeAll()`, `insert(key, value)`, `update(key, value)`, or `touch(key)`:
 - a. As part of the method call, T1 goes to the container server for this object or objects and gets an X lock on the objects involved. The X lock is held until T1 commits. Note that if another transaction T2 currently holds an S lock or U lock, the X lock request will fail.
 - b. T1 will block *M* seconds; *M* is configurable.
 - c. If T1 is unable to get the X lock, an exception is thrown to the client. It is for this reason that many people think (in an oversimplified but basically accurate way) that PESSIMISTIC means that an object is locked on `get(key)`, because once a transaction gets an object, no other transaction can change it.

4. T1 commits; either the client code calls `session.commit()` or the commit is done automatically by WebSphere eXtreme Scale:
 - a. T1 goes to the container server for this object or objects.
 - b. If it holds only an S lock for an object or objects, it releases the S lock. If it holds an X lock for an object or objects, T1 updates the object or objects in the BackingMap and releases the X lock.

OPTIMISTIC

The semantics for `lockStrategy=OPTIMISTIC` follow. Note that, for OPTIMISTIC locking, calls to the `get(key)`, `getAll()`, `getForUpdate(key)`, and `getAllForUpdate()` methods all behave the same.

1. Client application (transaction T1) calls the `get...` method:
 - If there is a near cache:
 - i. Transaction (T1) gets an S lock on the object or objects in the near cache. Note that if another transaction T2 is in the middle of a commit, T2 will hold an X lock on the object and thus the S lock request will fail.
 - ii. T1 will wait *M* seconds. If T1 unable to get the S lock, an exception is thrown to the client.
 - iii. If T1 gets the S lock, the object or objects are copied to the T1 transaction scoped cache ("T1 cache"), the OPTIMISTIC version value for this object is copied out of the object and saved, and the S lock is released. After this release, *no* lock is held by T1. The `get` method call returns to the client.
 - iv. If, during the previous steps, this object is not found in the near cache, T1 goes to the container server for the object. T1 gets an S lock at the container by the same logic as for the near cache, and any exceptions propagate back to the client. If T1 gets the S lock on the container server, the object and its version value are serialized back to the client near cache, the server-side S lock is released and the near cache logic continues (that is, the object or objects are copied to the T1 cache, the version value is copied, and the S lock is released).
 - If there is no near cache:
 - i. T1 goes to the container server for this object or objects and gets an S lock on the object or objects.
 - ii. Then, the same logic is followed as for the near cache, and any exceptions propagate back to the client.
2. Client (T1) reads or modifies the object data. Any changes are made only to the object in the T1 cache. No locks are obtained or checked.
3. Client (T1) does `put(key)`, `putAll()`, `remove(key)`, `removeAll()`, `insert(key, value)`, `update(key, value)`, or `touch(key)`. The operation is done on the object in the T1 cache. No locks are obtained or checked.
4. Client (T1) commits; either client code calls `session.commit()` or it is done automatically by WebSphere eXtreme Scale:
 - If there is a near cache:
 - i. If there is a near cache, T1 gets an X lock on the object or objects in the near cache. If it cannot get the X lock, T1 immediately returns to the client with an `OptimisticCollision` exception (perhaps inside another exception). It does not block, because if another transaction has an X lock, the OPTIMISTIC version field is being changed and any retry will fail anyway.

ii. Assuming no exception, T1 compares the OPTIMISTIC version field's current value with the value saved at "get..." time:

- If the version fields differ, T1 throws an `OptimisticCollision` exception (perhaps inside another exception), which indicates that another transaction changed the object since T1 got it and was working on it.
- If the version fields match, T1 updates the object in the near cache and releases the X lock.

If the operation being committed is not an update but to remove the object, the behavior is the same. Even if you did not do an explicit get prior to the remove, WebSphere eXtreme Scale does an implicit get during the "remove(...)" call to obtain the version field value. That value is used as described previously to decide if the remove is committed.

If during commit, the map no longer contains any such object, there is no "current" version field value to compare. In this case, the commit always succeeds with the incoming object inserted to the map.

- After dealing with the near cache (if there was one), T1 goes to the container or containers that host this object or objects and gets an X lock on the object or objects. The same behavior described in step 4 for the near cache is repeated at the container. Any exceptions are propagated back to the client.

NONE

The semantics for `lockStrategy=NONE` follow:

1. Client (T1) does a `get(key)` or any other get method:
 - If there is a near cache, no lock is obtained or checked for the object or objects. The object is copied from the near cache to the T1 cache in whatever state it is found (possibly with part but not all of its attributes modified if another transaction T2's commit is in progress).
 - If there is no near cache, T1 goes to the container server for this object or objects with the same logic (or lack thereof) and caveats as for the near cache.
2. Client (T1) reads or modifies the object data. Any changes are made only to the object in the T1 cache. No locks are obtained or checked.
3. Client (T1) does `put(key)`, `putAll()`, `remove(key)`, `removeAll()`, `insert(key, value)`, `update(key, value)`, or `touch(key)`. The operation is done on the object in the T1 cache. No locks are obtained or checked.
4. T1 commits; either client code calls `session.commit()` or it is done automatically by WebSphere eXtreme Scale. No locks are obtained or checked at any time:
 - If there is a near cache, T1 updates the object in the near cache regardless of what any other transaction might be doing (even if another transaction T2 is in the middle of its own commit).
 - After dealing with the near cache (if there was one), T1 goes to the container or containers for this object or objects with the same logic and caveats as for the near cache.

6.4 Serialization performance

Java serialization is the default mechanism for copying and transmitting objects over the network. Serialization is an expensive task and is something that is done each and every time that you do a PUT or GET operation with data into or out of an ObjectMap. Depending on the size and composition of your data object, serialization can be a real performance issue.

By default, WebSphere eXtreme Scale uses a Java serialization mechanism that can consume a significant amount of CPU. Tuning the serialization process can have a major positive performance impact. WebSphere eXtreme Scale provides options for managing the serialization process:

- ▶ Custom serialization with the Externalizable or Serializable interface

An application can add “implements Serializable” to its object, or it can have the objects implement the Externalizable interface, which can be many times faster. (Clients in actual client situations have observed seven to 10 times faster.)

- ▶ Implementation of the ObjectTransformer interface

You can use the ObjectTransformer plug-in in the following situations:

- You have non-serializable objects, and you do not have the source code access to enhance them.
- You want to further improve serialization performance for serializable or externalizable objects.
- You want to add custom code to copy a key or value for an entry.

The *serialize* and *inflate* methods of the ObjectTransformer allow an application to provide custom code to optimize these operations. The *serialize methods* serialize the object and provide a stream. The *inflate methods* provide the input stream and expect the application to create the object, inflate it using data in the stream, and then return the object. The implementations of the *serialize* and *inflate* methods must mirror each other.

If an application provides an ObjectTransformer for each BackingMap, the CPU cost of serialization can drop significantly.

If you experience performance problems that are related to serialization, the preferred practice is to provide ObjectTransformers for every map, if possible in combination with implementing Externalizable in the key and value classes.

6.4.1 A discussion of efficient serialization

Most distributed systems have a need at a certain point for objects to be moved from one JVM to another. The standard Java way to do this move is via serialization - converting a Java object to a byte array or stream, transmitting that stream of bytes over a TCP connection (of a specific form), and converting the stream back into the Java object. Two classic cases where serialization of Java objects is necessary is HTTP session replication and with WebSphere Extreme Scale used as a distributed cache. In Java, this serialization is managed through either the `java.lang.Serializable` or `java.lang.Externalizable` interface. `Serializable` is simpler to use, but it is extremely inefficient, which makes `Externalizable` the better choice. To reinforce this point, the default Java Developer Kit 1.6 serialization of a type String “A” is 8 bytes (not 2 bytes as you might expect), an Integer type is 81 bytes (not 2 to 4 bytes), and a GregorianCalendar type is 3112 bytes (when you typically only need to save `mmddyyyyhhmmss` and maybe a few bytes for zone and format if it is not Greenwich mean time (GMT)). You can do better.

Dealing with efficient serialization will not greatly complicate the programming model of an application or corporate application standard. The modest skill that is needed to write the two Externalizable methods can be learned quickly and only requires simple Java skill to begin.

The efficient serialization logic can be put in the Externalizable methods for use with other serialization-using functions, such as HttpSession replication. This efficient serialization logic can be put in the same Externalizable methods in the objects being stored in WebSphere eXtreme Scale, in a separate class implementing the WebSphere eXtreme Scale ObjectTransformer interface and attached to the WebSphere eXtreme Scale map, or split between them. The efficiency that you can gain is the same for any of these three (except as mentioned in the next paragraph).

The following article provides a good description of how serialization in Java works:

- “The Java serialization algorithm revealed”

<http://www.javaworld.com/community/node/2915>

This article does not mention Externalizable specifically, but it does give you an idea of the overhead that you can avoid by using both Externalizable and the WebSphere eXtreme Scale ObjectTransformer interface.

The following article describes Serialization more broadly, including Externalizable (note the paragraph about “Performance Considerations” near the end, which recommends that you “write your own protocol” to achieve the best performance, which is what we have explained how to do here):

- “Discover the secrets of the Java Serialization API”

<http://java.sun.com/developer/technicalArticles/Programming/serialization/>

It is not always necessary to have efficient serialization, although usually it is. Efficient serialization makes the operation of WebSphere eXtreme Scale faster for that object. Gets are faster, because there is less data to flow over the wire from the WebSphere eXtreme Scale container to the client application. Puts are faster for the reverse flow from the client to the container and the replica container. In both cases, far less CPU is used when the logic is efficient. However, if the logic using the object is not performance-sensitive (if it is used infrequently and if getting the object is one small part of an already long operation), you might not have to bother. Performance testing will let you know.

Existing packages for improved serialization with less writing code

There are packages on the Internet that can help you improve serialization. WSSUtils also includes serialization support. Neither of these packages is as efficient as writing your own code, but they require substantially less coding, because they both use modern introspection. You merely have to assert in one line that you want a given class to have serialization done. We have not tested either of these packages as of the time of publishing this book.

6.4.2 Implementing the Externalizable interface

When you implement the Externalizable interface, you are then required by the interface contract to provide implementations of the readExternal(java.io.ObjectInput in) and the writeExternal(java.io.ObjectOutput out) methods. You get complete control over the serialization process. See Example 6-8.

Example 6-8 Externalizable example

```
public class User implements Externalizable implements Cloneable {
    private static final long serialVersionUID = 1L;
```

```

private int userId;
private String firstName;
private String lastName;
private String emailAddress;
private long dateCreated;
private long dateModified;

// Getters and setters go here. Also a clone() implementation.

public void writeExternal(ObjectOutput out) throws IOException {
    out.writeInt(userId);
    out.writeUTF(firstName);
    out.writeUTF(lastName);
    out.writeUTF(emailAddress);
    out.writeLong(dateCreated);
    out.writeLong(dateModified);
}

public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
    this.userId = in.readInt();
    this.firstName = in.readUTF();
    this.lastName = in.readUTF();
    this.emailAddress= in.readUTF();
    this.dateCreated= in.readLong();
    this.dateModified= in.readLong();
}
}

```

Use Externalizable for efficient serialization

The previous section showed a fairly simple Java class implementing Externalizable. To show how a more complex class might be handled, we have included a heavily commented example Java class to demonstrate Externalizable methods so that you can see that the logic is not complex but follows a basic pattern. The class is named `FastSerializableKeyOrEntry_Externalizable.java` and is in Appendix A, “Sample code” on page 187.

Material for download: This file is also in the `SerializationUtils.jar` file that accompanies this book (see Appendix B, “Additional material” on page 197). This class makes use of the `SerializableUtils` class (also in the `.jar`).

If you examine this Java class, specifically the class definition, the instance variables, and the code in the `readExternal()` and `writeExternal()` methods, you can see that this code is not highly complex. After you learn the pattern, you merely follow it for each class and its fields.

You can go one step further in object serialization if you want to tackle dealing with separate object versions at the same time (a task that can often be more effort than it is worth). Begin your `writeExternal()` and `readExternal()` methods by writing or reading the `serialVersionUID` value. Do not generate a `serialVersionUID` but instead hardcode your own value, for example, start at 1 and increment each time that you change the fields in the object, which is a change that necessitates a change to `writeExternal` or `readExternal`. By writing the value, you are

recording the object version in the byte stream. When you read the value, you can compare it to your current (static final) `this.serialVersionUID` and therefore will know what you have to do differently in `readExternal()` to convert a stream of version X into an object of version Y. When you first write your objects and only version 1 exists, you can skip the comparison step; you can add it when you first create version 2.

The downloadable sample code also includes examples of other useful techniques that you might need, such as a custom `hashCode()` method.

Overriding `hashCode()` and `equals()`: Also important for WebSphere eXtreme Scale

The following article provides excellent high-level information about the topic of hashing:

- “Java theory and practice: Hashing it out”

<http://www.ibm.com/developerworks/java/library/j-jtp05273/index.html>

Overriding the `hashCode()` and `equals()` methods can yield a significant performance boost for WebSphere eXtreme Scale key and value objects (`equals()` is important for value objects, `hashCode()` not so much).

The `hashCode()` method on your key object is also central to WebSphere eXtreme Scale determining into which partition (and thus which container JVM) that key and its value will be stored. WebSphere eXtreme Scale uses the `hashCode()` value from the key object and then does modulo arithmetic with the `numberOfPartitions` value to produce a partition number from 0 to `numberOfPartitions-1`.

If, for any reason, you want to directly control to which partition each key-value pair goes (the default works well in most cases), strongly consider using `com.ibm.websphere.objectgrid.plugins.PartitionableKey`. This class lets you control partitioning, but it does not affect other uses of `hashCode()`, such as the even distribution of objects in the internal WebSphere eXtreme Scale HashMaps, which is a good thing for overall performance.

The ideal implementation has `MyKey.hashCode()` produce a good random value, but it might have `PartitionableKey`'s method return a separate value that might not be as evenly distributed over all possible int values (as `hashCode()` ideally returns) but will achieve the partitioning that you want.

6.4.3 Using a custom `ObjectTransformer` implementation

Think back to the article about serialization that we referenced previously (“The Java serialization algorithm revealed” at <http://www.javaworld.com/community/node/2915>) and our own `Externalizable` preferred practices. The overhead mentioned there is avoided for embedded objects in the “root” WebSphere eXtreme Scale-stored class, because the root class's `Externalizable` methods directly call the `externalizable` methods of those embedded classes (and those do the same for their embedded classes, recursively). The WebSphere eXtreme Scale `ObjectTransformer` class can be used, if necessary, to avoid the overhead for the root class itself.

If a `BackingMap` has an `ObjectTransformer` plug-in configured on it, WebSphere eXtreme Scale will call methods of the transformer for serialization and inflation, and it will do so instead of calling the corresponding `Serializable/Externalizable` methods. To keep good encapsulation, keep the serialization code for your object's fields in the object. That is, your `ObjectTransformer` must immediately (and only) call your object class's `externalizable` methods (`readExternal(...)` or `writeExternal(...)`). If you do not have the ability to make the

object classes implement `Externalizable` or achieve efficient serialization in another way, you can resort to putting the full serialization and inflation logic in the `ObjectTransformer` class. The `ObjectTransformer` interface has methods for serializing and inflating both keys and values.

The `ObjectTransformer` interface also has two methods for a related purpose: `copyKey` and `copyValue`. The default `copyKey` and `copyValue` method implementations first attempt to use the `clone()` method, if provided in the object class. If no `clone()` method implementation is provided, the implementation defaults to serialization. However, you can often achieve better performance by writing intelligent code in a custom `ObjectTransformer`'s `copyKey` and `copyValue` methods. Remember to handle the situation where the value to be copied in `copyValue` is null; WebSphere eXtreme Scale does not permit null keys so your `copyKey` does not have to check this value.

As we have mentioned before, the default way that WebSphere eXtreme Scale copies objects is serialization, which includes calling any `ObjectTransformer` that is configured. Object serialization is also used directly when the `ObjectGrid` is running in distributed mode. The `LogSequence` uses the `ObjectTransformer` plug-in to help it serialize keys and values before transmitting the changes to peers in the `ObjectGrid`.

The following list details how `ObjectGrid` tries to serialize both keys and values:

- ▶ If a custom `ObjectTransformer` plug-in is written and plugged in, `ObjectGrid` calls methods in the `ObjectTransformer` methods to serialize keys and values and get copies of object keys and values.
- ▶ If a custom `ObjectTransformer` plug-in is not used, `ObjectGrid` serializes and deserializes according to the default. If the default is used, each object must be implemented as `externalizable` or as `serializable`:
 - If the object supports the `Externalizable` interface, the `writeExternal` and `readExternal` methods are called. Objects that are implemented as `externalizable` lead to better performance.
 - If the object supports the `Serializable` interface, the `writeObject` and `readObject` methods are called.
 - If the object does not support either of these methods, the default behavior is to throw a `serialization Exception`.

Writing an ObjectTransformer

An `ObjectTransformer` must implement the `ObjectTransformer` interface and follow the common `ObjectGrid` plug-in conventions. Example 6-9 shows the `ObjectTransformer` interface.

Example 6-9 ObjectTransformer interface

```
public interface ObjectTransformer {
    void serializeKey(Object key, ObjectOutputStream stream) throws IOException;
    void serializeValue(Object value, ObjectOutputStream stream) throws
IOException;
    Object inflateKey(ObjectInputStream stream) throws IOException,
ClassNotFoundException;
    Object inflateValue(ObjectInputStream stream) throws IOException,
ClassNotFoundException;
    Object copyValue(Object value);
    Object copyKey(Object key);
}
```

Using ObjectTransformer

You have two approaches to add an ObjectTransformer into the BackingMap configuration. We describe both approaches in this section.

Programmatically plug in an ObjectTransformer

Assume that the class name of the ObjectTransformer implementation is the `com.company.org.MyObjectTransformer` class. This class implements the ObjectTransformer interface. Example 6-10 creates the custom ObjectTransformer and adds it to a BackingMap.

Example 6-10 Creating the custom ObjectTransformer

```
ObjectGridManager objectGridManager =
ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("Grid", false);
BackingMap backingMap = myGrid.getMap("User");
MyObjectTransformer myObjectTransformer = new MyObjectTransformer();
backingMap.setObjectTransformer(myObjectTransformer);
```

XML configuration approach to plug in an ObjectTransformer

You can also configure an ObjectTransformer using XML. The XML that is shown in Example 6-11 creates a configuration that is equivalent to the described program-created ObjectTransformer.

Example 6-11 Creating an XML configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="Grid">
      <backingMap name="User" pluginCollectionRef="User" />
    </objectGrid>
  </objectGrids>

  <backingMapPluginCollections>
    <backingMapPluginCollection id="User">
      <bean id="ObjectTransformer"
className="com.company.org.MyObjectTransformer" />
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

Custom ObjectTransformer User class

You can write a custom object transformer class for the User class, from Example 6-8 on page 128. Example 6-12 shows the custom object transformer.

Example 6-12 Writing a custom object transformer

```
public class UserObjectTransformer implements ObjectTransformer {

    @Override
    public Object copyKey(Object key) {
        int userid =(Integer) key;
        int useridCopy= new Integer (userid);
```

```

        return useridCopy;
    }

    @Override
    public Object copyValue(Object value) {
        User user = (User) value;
        try {
            if (user != null) {
                return user.clone();
            }
        } catch (CloneNotSupportedException e) {
            // display exception message
        }
        return null;
    }

    @Override
    public Object inflateKey(ObjectInputStream stream) throws IOException,
        ClassNotFoundException {
        int userid = stream.readInt();
        return userid;
    }

    @Override
    public Object inflateValue(ObjectInputStream stream) throws IOException,
        ClassNotFoundException {
        User user = new User();
        //This assumes that null values are not possible. If they are, you need more logic
        user.readExternal(stream);
        return user;
    }

    @Override
    public void serializeKey(Object key, ObjectOutputStream stream)
        throws IOException {
        int userId = (Integer) key;
        stream.writeInt(userId);
    }

    @Override
    public void serializeValue(Object value, ObjectOutputStream stream)
        throws IOException {
        User user = (User) value;
        // This assumes that null values are not possible. If they are, you need more
        logic
        user.writeExternal(out);
    }
}

```

6.5 Query performance tuning

If you are beginning with WebSphere eXtreme Scale, you might rely too heavily on the query capability of WebSphere eXtreme Scale, especially when incorporating it into an existing application that is tailored to database access. WebSphere eXtreme Scale is primarily a caching provider. The query functionality is a valid convenience feature of the product, but it is not backed by all the sophisticated query optimizers that are found in top-end databases. Therefore, WebSphere eXtreme Scale is most efficient when finding an object by its key rather than when locating that same object by a query.

If you have to use query in WebSphere eXtreme Scale, try to use an index (or, better, the reverse index pattern) and avoid parallel queries where possible. If all partitions are touched, queries can only be as fast as the slowest individual system, which effectively eliminates scaling. To tune the performance of your queries, use the techniques and tips that we describe in this section.

6.5.1 Using parameters

When a query runs, the query string must be parsed and a plan must be developed to run the query, both of which can be costly. WebSphere eXtreme Scale caches query plans by the query string. Because the cache is a finite size, it is important to reuse query strings whenever possible.

Using named or positional parameters also helps the performance by fostering query plan reuse, as shown in the following example:

```
Query q = em.createQuery("select u from User where u.userId=?1");
q.setParameter(1, 123);
```

6.5.2 Using indexes

Proper indexing on a map might have a significant positive impact on query performance, even though indexing has overhead on overall map performance. Without indexing on the object attributes that are involved in queries, the query engine performs a table scan for each attribute. The table scan is the most expensive operation during a query run.

Indexing on object attributes that are involved in queries allows the query engine to avoid an unnecessary table scan, improving the overall query performance. If the application is designed to use query intensively on a read-most map, configure indexes for object attributes that are involved in the query. If the map is mostly updated, you must balance between query performance improvement and indexing overhead on the map.

When POJOs are stored in a map, proper indexing can avoid a Java reflection. In the following example query, the WHERE clause uses a range index search if the user ID field has an index built over it. Otherwise, the query scans the entire map and evaluates the WHERE clause by first getting the user ID using Java reflection and then comparing the user ID with the value 500.

```
SELECT u FROM User u WHERE u.userId < 500
```

Indexes have the following requirements when used by query:

- ▶ All indexes must use the built-in HashIndex plug-in.
- ▶ All indexes must be statically defined. Dynamic indexes are not supported.
- ▶ The @Index annotation can be used to automatically create static HashIndex plug-ins.
- ▶ All single-attribute indexes must have the RangeIndex property set to true.

- ▶ All composite indexes must have the `RangeIndex` property set to `false`.
- ▶ All association (relationship) indexes must have the `RangeIndex` property set to `false`.

An alternative to indexes with much better scalability and performance (at the price of having custom code) is the reverse index pattern, which is described in 5.4.4, “Indexing plug-ins” on page 107.

6.5.3 Using pagination

In client-server environments, the query engine transports the entire result map to the client. The data that is returned must be divided into reasonable chunks. The `EntityManager Query` and `ObjectMap ObjectQuery` interfaces both support the `setFirstResult` and `setMaxResults` methods that allow the query to return a subset of the results.

6.5.4 Returning primitive values instead of entities

With the `EntityManager Query` API, entities are returned as query parameters. The query engine currently returns the keys for these entities to the client. When the client iterates over these entities using the `Iterator` from the `getResultIterator` method, each entity is automatically inflated and managed as though it were created with the `find` method on the `EntityManager` built from the entity `ObjectMap` on the client. The entity value attributes and any related entities are eagerly resolved.

To avoid building the costly graph, modify the query to return the individual attributes with path navigation, for example:

```
// Returns an entity
SELECT u FROM User u
// Returns attributes
SELECT u.userId, u.firstName, u.lastName, FROM User u
```

6.5.5 Query plan

All WebSphere eXtreme Scale queries have a query plan. The plan describes how the query engine interacts with object maps and indexes. Display the query plan to determine if the query string or indexes are being used appropriately. You can also use the query plan to explore the differences that subtle changes in a query string make in the way that WebSphere eXtreme Scale runs a query.

The query plan can be viewed in one of two ways:

- ▶ `EntityManager Query` or `ObjectQuery` `getPlan` API method
- ▶ `ObjectGrid` diagnostic trace

6.5.6 getPlan method

The `getPlan` method on the `ObjectQuery` and `Query` interfaces returns a string that describes the query plan. This string can be displayed to standard output or a log to display a query plan.

Distributed environment note: In a distributed environment, the `getPlan` method does not run against the server and does not reflect any defined indexes. To view the plan, use an agent to view the plan on the server.

6.5.7 Query plan trace

You can display the query plan using ObjectGrid trace. To enable query plan trace, use the following trace specification:

```
QueryEnginePlan=debug=enabled
```

6.5.8 Query plan examples

The following query retrieves all account descriptions from a particular user by sequentially scanning the UserAccount map to get a UserAccount object. From the UserAccount object, the query navigates to its user object and applies the d.no=1 filter. In this example, each UserAccount has only one user object reference, so the inner loop runs one time:

```
SELECT e.description FROM UserAccount e JOIN e.User d WHERE d.userId=1
Plan trace:
for q2 in UserAccount ObjectMap using INDEX SCAN
for q3 in q2.User
filter ( q3.getUserId() = 1 )
returning new Tuple( q2.description )
```

The following query is equivalent to the previous query. However, the following query performs better, because it first narrows down the result to one user object using the unique index that is defined over the User primary key field number. From the user object, the query navigates to its user account objects to get the description:

```
SELECT e.description FROM User d JOIN d.UserAccount e WHERE d.userId=1
Plan trace:
for q2 in User ObjectMap using UNIQUE INDEX key=(1)
for q3 in q2.getUserAccounts()
returning new Tuple( q3.description )
```



Operations and monitoring

This chapter describes how to manage a WebSphere eXtreme Scale environment at run time. We assume that you already have a WebSphere eXtreme Scale topology in place, together with one or more applications. We also describe operations and monitoring in a production environment.

This chapter includes the following topics:

- ▶ Starting and stopping WebSphere eXtreme Scale
- ▶ The placement service in WebSphere eXtreme Scale
- ▶ The xsadmin command-line tool
- ▶ Configuring failure detection
- ▶ Monitoring WebSphere eXtreme Scale
- ▶ Applying product updates

7.1 Starting and stopping WebSphere eXtreme Scale

High availability and redundancy is inherently built into the WebSphere eXtreme Scale product. The product can tolerate a dynamic number of container server Java virtual machines (JVMs) starting and stopping in the environment and is built to withstand the catalog server cluster becoming partially available at any moment in time. Because of these qualities of service, application clients can continue to operate successfully with only a fraction of the grid running, so long as there is enough running capacity available for the incoming load and the primary data. These capabilities also allow for client applications to operate interruption-free during maintenance cycles for hardware, operating systems, and even the WebSphere eXtreme Scale run time without having to alter the application client, as long as enough of the grid remains available.

There are times, however, where a full grid outage must occur, for example, if a change is made to the size of the catalog server cluster, a change in the number of grid partitions or to the grid definition, or a change in the application logic running in the grid that cannot tolerate a mixed version environment. A common way to handle this situation is to have a separate backup grid (“cold” or “hot”) with an equivalent configuration. If you have no such backup grid and if there are no provisions for applications to access a backup data storage tier of another kind, an application interruption will occur during a full grid outage.

A WebSphere eXtreme Scale environment must be stopped and started in an orderly manner, as illustrated in Figure 7-1.

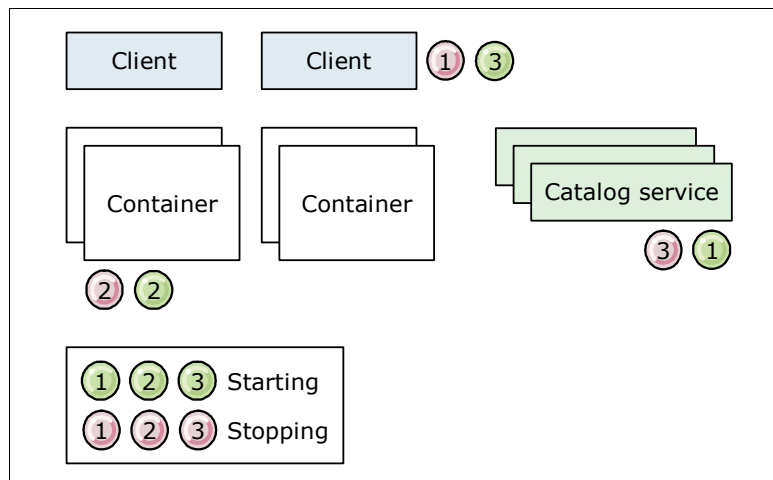


Figure 7-1 Order for stopping and starting the WebSphere eXtreme Scale environment

When starting the environment, start the catalog servers first, then the container servers, and finally, the clients. When stopping, the order is reversed. Stop the clients, then the container servers, and finally the catalog servers.

You can obtain information about stopping and starting WebSphere eXtreme Scale processes in the WebSphere eXtreme Scale Information Center:

- ▶ Starting and stopping stand-alone servers:
<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.e xtremescale.admin.doc/txssastartstop.html>
- ▶ Starting and stopping servers in a WebSphere Application Server environment:
<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.e xtremescale.admin.doc/txscatalogstartwas.html>

- Using the embedded server API to start and stop servers:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.e xtremescale.admin.doc/txsadminapi.html>

This section will cover the common scenarios.

7.1.1 Starting and stopping catalog services

For high availability, production environments need to have a cluster of catalog servers (meaning, catalog JVMs) forming a catalog service domain, with each catalog server residing in a separate machine. The catalog servers must be started before the container servers (container JVMs), and all the catalog server cluster members must be started in parallel. Each of them waits for the others to join the core group. If one is not started, others will time out during the start attempt.

This section provides a quick overview of the methods that are used to start and stop catalog servers.

Starting a catalog service domain on stand-alone JVMs

A catalog service domain running on stand-alone JVMs is started with the **startOgServer** command in the *wxs_root/bin* directory. Example 7-1 shows the command and its options.

Example 7-1 startOgServer command for catalog services

```
startOgServer.sh <server_name> [with the following options]
  -catalogServiceEndpoints <server:host:clientPort:peerPort,...>
  -quorum true|false
  -heartbeat 0|1|-1
  -clusterSecurityFile <cluster security xml file>
  -clusterSecurityUrl <cluster security xml URL>
  -domain <domain_name>
  -listenerHost <hostname>
  -listenerPort <port>
  -serverProps <server properties file>
  -JMXServicePort <port>
  -traceSpec <trace specification>
  -traceFile <trace file>
  -timeout <seconds>
  -script <script file>
  -jvmArgs <JVM arguments>
```

No options are required when starting a single catalog server.

When starting multiple catalog servers in a catalog service domain, you need to specify a list of the catalog service endpoints for all the catalog servers in the domain (**-catalogServiceEndpoints** option), and the listener port (**-listenerPort** option) if you are not taking the default port number 2809. See 3.5.1, “Catalog service domain ports” on page 43 for more information and an example.

A catalog service domain name (**-domain** option) is not required when starting a catalog service. However, if you are using multi-master replication or are using multiple catalog service domains within the same set of processes, you need to define a unique catalog service domain name. The default domain name is `DefaultDomain`.

For more information about starting the catalog service domain in a stand-alone JVM environment, see “Starting a stand-alone catalog service” at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r0/topic/com.ibm.websphere.extramescale.admin.doc/txscatalogstart.html>

Stopping catalog servers in stand-alone JVMs

Run the **stopOgServer** script to stop one or more catalog servers. Example 7-2 shows the command and its options.

Example 7-2 stopOgServer command for catalog service

```
stopOgServer.sh <serverName[, servername]> -catalogServiceEndPoints
<host:ORBport,host:ORBport> [with the following options]
    -traceSpec <trace specification>
    -traceFile <trace file>
    -clientSecurityFile <security properties file>
```

Note that the **-catalogServiceEndPoints** option for stopping a catalog service differs from the same option that is used to start the service. When starting a catalog service, this option specifies the peer and client access ports. When stopping a catalog service, the option specifies the Object Request Broker (ORB) ports, because, as for starting and stopping containers, you stop a catalog server by communicating to the already running catalog service, which you do by talking to the ORB port.

Starting and stopping catalog services in WebSphere Application Server

You can configure the catalog service to run on multiple processes in WebSphere Application Server, including the deployment manager, node agents, and application servers. The catalog service will start and stop when the WebSphere Application Server processes start and stop.

7.1.2 Starting and stopping container servers

This section provides a quick overview of the methods that are used to start and stop container servers.

Starting a container server on a stand-alone JVM

To start a container server on a stand-alone JVM, you use the **startOgServer** command. You need to specify the ObjectGrid XML file (see 5.1.1, “Server XML configuration” on page 71). You must also specify the ORB host and port of the catalog service with the **-catalogServiceEndPoints** option. The host and port pairs that are specified for this option when you start the container server must match the values of the **-listenerHost** and **-listenerPort** values that were specified for the catalog server when it was started.

Example 7-3 shows the **startOgServer** command and its options for starting container servers.

Example 7-3 startOgServer command for container servers

```
startOgServer.sh <server_name> -objectgridFile <xml file>
| -objectgridUrl <XML URL> [with the following options]
  -catalogServiceEndPoints <host:ORBport,...>
  -deploymentPolicyFile <deployment policy XML file>
  -deploymentPolicyUrl <deployment policy XML URL>
  -listenerHost <hostname>
  -listenerPort <port>
  -serverProps <server properties file>
  -JMXServicePort <port>
  -traceSpec <trace specification>
  -traceFile <trace file>
  -timeout <seconds>
  -script <script file>
  -jvmArgs <JVM arguments>
```

It is recommended that you specify a deployment policy file to set up partitioning and replication. The deployment policy can also be used to influence placement behavior. If you do not specify a deployment policy file, the default is used with regard to replication, partitioning, and placement. These values are insufficient for most installations.

Stopping a container server on a stand-alone JVM

To stop the container servers, you can use the **stopOgServer** script or the **xsadmin -teardown** command. You can use either one to stop one container or a list of containers. The **stopOgServer** script stops the container processing logic and also the JVM in which the container was running. The **xsadmin -teardown** command stops the container processing logic, but it does not stop the JVM.

Example 7-4 shows the **stopOgServer** command and its options.

Example 7-4 stopOgServer command for catalog service

```
stopOgServer.sh <serverName[, servername]> -catalogServiceEndPoints
<host:ORBport,host:ORBport> [with the following options]
  -traceSpec <trace specification>
  -traceFile <trace file>
  -clientSecurityFile <security properties file>
```

The **-catalogServiceEndPoints** option contains the list of catalog server hosts and ORB ports, which is the same list that you used to start the container server.

We describe the **xsadmin -teardown** command further in “Using the xsadmin -teardown command” on page 145.

Starting and stopping a container server in WebSphere Application Server

The container server is tied to the application. When you start or stop the configured application, the container servers also start and stop. You can also use the **xsadmin -teardown** command or the **tearDownServers** operation on the **PlacementServiceMBean** to stop the grid.

For more information about the PlacementServiceMBean, see this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.websphere.extremescale.javadoc.doc/topics/com/ibm/websphere/objectgrid/management/PlacementServiceMBean.html>

7.1.3 Performing a full (cold) start of WebSphere eXtreme Scale

A cold start of the grid starts up in the following order:

1. Start the full set of catalog servers and ensure that they are active and in good health.

The catalog service can run in a single process, or it can include multiple catalog servers in a cluster to form the catalog service domain. (Remember, for production, always have multiple catalog servers.) You must start all the catalog servers in a catalog service domain at the same time, because each catalog server waits for the other catalog servers to join the core group before starting. A catalog server will eventually time out if no other servers become available.

If quorum is enabled, you can inspect catalog server health by checking the quorum status.

For information about enabling quorum, see 7.2.2, “Quorum and the placement service” on page 150.

For information on checking the quorum status, see “Managing data center failures” at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.websphere.extremescale.admin.doc/txsquorcatsrovr.html>

2. Start the container servers.

You do not have to start all the container servers before allowing shard placement to occur, but it is usually best to do so. Consider that each time that a container server starts, it can trigger a rebalance of unplaced and previously placed shards, creating a load on processors and the network.

To reduce placement service churn in large topologies, set the `numInitialContainers` property in the deployment policy descriptor XML file to the full number of containers that you normally start. Setting this property will halt the placement service from generating and implementing placement plans until the specified number of containers are online. For more information about the `numInitialContainers` property, see “Deployment policy descriptor XML file” at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extremescale.admin.doc/rxsdplcyref.html>

With WebSphere eXtreme Scale V7.1.0.2 or later, you can use the **xsadmin -triggerPlacement** command to temporarily override the `numInitialContainers` value in special cases, for example, when you perform maintenance on your servers and want shard placement to continue running.

It is a preferred practice to start small groups of container servers at a time so that you do not overload processor capacity. Experiment with your particular hardware to discover the optimal number. If large numbers of container servers are being concurrently started, the catalog server ORB must be properly tuned to have enough threads to handle the concurrent starts and there must be enough CPU resource on the catalog server host. If your containers are managed (they are WebSphere Application Server servers) and you have WebSphere Application Server V7 or higher, starting the entire cluster of containers will result in WebSphere Application Server starting each member of the cluster one at a time. This approach tends to work well, because you avoid excessive CPU and thread

use, with the total time to start all the containers often being less than if you tried to start them in groups (even small groups).

For more information, see this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.websphere.extremescale.admin.doc/rxsorbproperties.html>

3. Start the clients.

When the container servers are active, the grid is available for the clients.

7.1.4 Performing a partial start of container servers

Subsets of the grid can be started in batches of multiple containers by using the WebSphere Application Server cluster start, or with the WebSphere administrative console. If the `numInitialContainers` configuration is being used, placement will not occur until that threshold is reached, or `triggerPlacement` on the `PlacementServiceMBean` is invoked.

You can also use the `xsadmin -suspendBalancing` and `xsadmin -resumeBalancing` commands (or the `PlacementServiceMBean`) to suspend and resume the balancing mechanism of the placement service. This action can help reduce CPU utilization when “bouncing” a server in a topology where there are only a few servers total. If you have two servers and stop one, there will be no replicas anymore. When you restart the first container on the second server, that container will be flooded with demands to host replicas. By running `xsadmin -suspendBalancing` before stopping the second server and waiting until after all containers on the second server are restarted before you run `xsadmin -resumeBalancing`, replicas will be spread evenly across all the containers. Even if you have more than two servers, suspend and resume will avoid wasting time replacing replicas multiple times when you are perform a quick “bounce” of a server.

See 7.3.2, “Useful xsadmin commands” on page 157 for more information about these commands.

Verifying that a container server start was successful

When start-up occurs, the placement service is responsible for generating a new placement plan for placing primary and replica shards onto the new containers. To ensure that the placement service has completed its work for all the new containers, you can check the placement status to verify that there are no outstanding work items (see “`xsadmin -placementStatus`” on page 158).

After the placement service has finished, the `xsadmin -containers` command retrieves the catalog server’s view of the current grid placement. Using the same filter parameters that were described with the `teardown` command (see “`xsadmin -teardown`” on page 160), an operator can get a view of the number of container servers that are running on a host, zone, or the entire grid.

Example 7-5 Commands to view the current grid placement

```
xsadmin -containers
xsadmin -containers -fz <zone_name>
xsadmin -containers -fh <host_name>
```

You can use the current catalog server view of the running topology to confirm that the number of container servers that you expect to be available matches the catalog server’s placement view.

The `xsadmin -routetable` command (“xsadmin -routetable” on page 159) simulates a new client connection to the data grid and displays each partition state (reachable, unreachable, and invalid) for the grids. This command can be invoked after the placement service’s work has been completed. If no partitions are marked as unreachable, the grid is considered healthy.

7.1.5 Performing a partial stop of catalog servers

When you run with multiple catalog servers, one catalog server is the master (primary) and the other catalog servers serve as replicas. If the primary is stopped, another active catalog server will be elected to become the new primary. If a partial grid stop affects all but one of the catalog servers, it is best to minimize this downtime, because the remaining catalog server will not have a backup in place if a failure occurs. The leading practice is to make sure that at least two catalog servers are running and synchronized at all times, even during a partial grid stop.

Additionally, when performing a rolling restart of catalog server JVMs, it is important to ensure that a newly restarted catalog server has replicated with the primary before stopping the primary catalog server. Currently, there is no way to tell when a new catalog server JVM is fully synchronized with the primary. Therefore, it is best to wait a minimum of 5 minutes between a start of a replica catalog server and the stop of the primary catalog server.

Note that when a catalog server is stopped by using `stopOgServer`, the quorum drops to one less server. Therefore, the remaining servers still have quorum. Restarting the catalog server bumps quorum back to the previous number.

7.1.6 Performing a partial stop of container servers

Stopping only part of the container servers in the grid can provide you with the opportunity to perform hardware or operating system maintenance, while still allowing client applications to continue to operate without interruption. Of course, that is assuming that there is still enough capacity (memory and CPU) on the remaining grid to service the incoming demand.

If a partial grid stop is necessary, it is important to understand the failover capabilities of the remaining grid infrastructure. If zones are configured for replica placement and *you have only two zones*, stopping a zone can also stop all grid replicas. When hardware or operating system maintenance occurs, it is preferable to keep the grid running across two physical machines (best) or two virtual machines and be aware of the zone placement. If it is necessary to consolidate the grid down to one virtual or physical machine or zone, this window needs to be minimized to reduce the window of time that the grid is susceptible to hardware, operating system, or JVM failures.

Using development mode

Unless in *development mode*, replica and primary shards will not be placed on containers that have the same IP address. When in development mode, replicas are placed without regard to IP address, zone, or any other rules. You must only use development mode, as you might guess, during development on a single workstation and when you want to see replicas created and used. Because the existence of replicas has no effect on application behavior (only on performance), development mode is seldom useful at all.

Development mode is set to `true` by default in the deployment policy descriptor XML file. This choice was made because clients experimenting with eXtreme Scale on their workstations frequently called up asking, “Where are my replicas?”. Now, you understand how this process works. The preferred practice (unless you have a special need to act otherwise) is that all

descriptor XML files must explicitly set development mode to false. You can obtain information about development mode at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.websphere.extremescale.admin.doc/rxsdplcyref.html>

You can obtain information about shard placement with zones at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extremescale.admin.doc/txsplshzones.html>

Using the **xsadmin -teardown** command

When a container server stops, several steps occur to *tear down* the container and ensure that new primary shards are elected and that replicas are created. These promotion and move decisions are made by the placement service that runs in the primary catalog server. When the **stopOgServer** script is invoked multiple times in succession, the placement service treats each stop individually and can cause excess processing while developing new placement strategies. It is a preferred practice to batch container server stops in one of three ways.

The first way is to use **stopOgServer** and pass a list of container servers, allowing time for these servers to stop before stopping the next batch. Another way is by using the **xsadmin -teardown** command with a list of servers or filter parameters for host or zone (which **stopOgServer** does not have).

Yet another way is to use the **xsadmin -suspendBalancing** command, then **stopOgServer** with either a list of servers or multiple **stopOgServer** commands with one server each, and then use **xsadmin -resumeBalancing** when all desired servers are stopped. The **-teardown** option does not always stop the JVM itself, where **stopOgServer** will do so.

You can use either technique to easily stop all containers or catalog servers for a zone, host, or specific list of servers.

For information about using the **xsadmin -suspendBalancing** and **-resumeBalancing** commands, see this website:

<http://www-01.ibm.com/support/docview.wss?uid=swg1PM37059>

For information about using the **xsadmin -teardown** command, see this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extremescale.admin.doc/txsteardown.html>

Before performing a batch teardown of container servers (or performing any maintenance on a container server), verify the health of the master catalog server. Use monitoring tools or directly check the log and first-failure data capture (FFDC) files for exceptions, such as `LockTimeoutExceptions`. If the health of the catalog server is suspect, perform a recycle of the catalog server before performing container maintenance.

When running in WebSphere Application Server: A teardown will bring down all WebSphere eXtreme Scale runtime components in the application server, but it will not stop the application server process.

Furthermore, after a teardown has completed, it is not possible to relaunch containers or start new containers in those WebSphere Application Server processes until they have been restarted. After invoking the **teardown** command, stop the processes by using the normal administration mechanisms. Or, you can use **xsadmin -suspendBalancing** and **-resumeBalancing** and stop the servers in the normal way with WebSphere Application Server.

Tearing down container servers

You can use the **xsadmin -teardown** command to stop a list or group of container servers. You can provide a list of servers after the **-teardown** parameter:

```
xsadmin -teardown <container_server_name>[,<container_server_name>]
```

Tearing down an entire zone

Use the **-fz** parameter and provide the name of the zone. The catalog server determines the servers that are running in the zone, and the **xsadmin** tool prompts you with a list of the servers in the selected zone before shutting down the servers:

```
xsadmin -teardown -fz <zone_name>
```

Tearing down all servers on a particular host

Use the **-fh** parameter and provide the name of the host. For example, to shut down all the servers on myhost.mycompany.com, enter **-fh myhost.mycompany.com**. The catalog server determines the servers that are running on the host, and the **xsadmin** tool prompts you with a list of the servers in the selected host before shutting down the servers:

```
xsadmin -teardown -fh <host_name>
```

Verifying that a container server teardown was successful

The **xsadmin -containers** command retrieves the catalog server's view of the current grid placement. Using the same filter parameters that were described with the **teardown** command, an operator can get a view of the number of container servers that will be affected by tearing down all containers on a host, zone, or entire grid:

- ▶ **xsadmin -containers -fz <zone_name>**
- ▶ **xsadmin -containers -fh <host_name>**

The current catalog server view of the running topology provides a good baseline before performing a batch teardown in the environment. You can use this view, coupled with the knowledge of the number of containers that are expected to be affected during a batch teardown, to confirm the **xsadmin -containers** output after the teardown operations.

By invoking the **xsadmin -containers** command after the batch teardown, you can confirm that the placement service has acknowledged and made the appropriate plan changes to reflect the physical topology change. Note that the actual changes occurring on the containers might still be in-flight even if the catalog server's view appears complete.

When a batch teardown occurs, the placement service is responsible for generating a new placement plan for promoting primaries and moving replica shards onto the remaining containers. To ensure that the placement service has completed its work, inspect the

numOutStandingWorkItems in the placement status to make sure that it has drained to zero (See “*xsadmin -placementStatus*” on page 158).

The **xsadmin -routetable** command simulates a new client connection to the data grid and displays each partition state (reachable, unreachable) for the grids. You can invoke this command after the placement service’s work completes. By inspecting the output for partitions that are marked as unreachable, you can confirm the health of the grid.

7.1.7 Performing a full stop of WebSphere eXtreme Scale

When it is necessary to perform a full grid stop, an application interruption of service will occur unless the application has been instrumented with logic to switch onto another means of data storage, such as a database, while the grid is offline.

The catalog servers must be the last servers to stop in the grid environment during a graceful shutdown. If it is necessary to force a termination of the entire grid (by killing the Java process), the catalog servers must be the first processes terminated so that they do not attempt to heartbeat and take failover actions while the grid comes down. Follow this sequence:

1. Quiesce the traffic in the grid.
2. Verify that the traffic has been drained from the grid.
3. Stop the grid.
4. Stop the catalog servers.

Grid availability and quiesce

For transactions that are short-lived (less than 30 seconds), WebSphere eXtreme Scale client applications can leverage the ObjectGrid quiesce capabilities. These capabilities allow a grid to be left running, but to move through ONLINE → QUIESCE → OFFLINE states for draining existing traffic and blocking all new traffic. You must create an application client to invoke an API to manipulate the state of the grid, and applications must tolerate the grid in the OFFLINE state.

For more information, see this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.webSphere.extremescale.admin.doc/txssetavail.html>

Verifying that the traffic has been drained from the grid

If the data that is being stored in the grid is session-related data, or if it has a short life span, you can monitor the count of entries in the grid’s maps (using **xsadmin -mapsizes** or other tools) until it drains down to zero.

If the data is long-lived, you can inspect the grid’s transaction count metric to ensure that it has fallen flat. You can view the metric using a custom Java Management Extensions (JMX) client or a third-party monitoring tool to monitor grid throughput. (See 7.5.6, “Additional monitoring tools” on page 181.)

Stopping the entire grid

The **xsadmin** utility’s **teardown** command without any filter parameters will tear down all servers in the grid. You can also use **stopOgServer** and specify a list of servers that includes all servers in the grid. Because eXtreme Scale is designed to be robust and handle the sudden failure of any or all servers in the grid with no problem restarting later, you can also just kill all processes related to the grid.

Teardown failed, now what

Check the logs of the primary catalog server to ensure that the catalog server is in a good state with no current exceptions. If the catalog server appears to be in an unhealthy state, yet the replica catalog servers appear healthy, stop or kill the process of the primary catalog server and invoke the teardown again.

If the teardown was invoked using filtering options and the filtering appears to have failed, attempt to invoke the teardown by passing in an explicit list of container servers to tear down.

If the teardown status for a server has failed, the best course of action is to take a series of thread dumps from the process and then force termination (kill). Save all thread dumps and logs from the process for evaluation. As long as the server is not listed in the output from the **xsadmin -containers** command, it is not considered available from the perspective of the catalog server's placement plans and routing. If the server still appears in the **xsadmin -routetable** or **xsadmin -containers** output, attempt another **teardown** command action.

If all else fails, you can kill any processes that are in an unhealthy state; the ability to restart a catalog or container will not be harmed by killing its process. If the processes that you kill include all catalog servers or so many containers that primary data is lost, the grid is effectively down and all remaining processes must be stopped or killed. The grid can then be restarted.

7.1.8 What to do when a JVM is lost

You manage high availability in WebSphere eXtreme Scale through the use of core groups and the high availability manager. The catalog service is responsible for placing container servers in core groups. A single member of the core group is designated as the leader of the core group. The core group leader contacts the JVMs in the group and reports failures and membership changes (a failed or new JVM) to the catalog service. JVM failures are detected by the core group leader through missed heartbeats or from recognizing that a JVM's socket has closed. The catalog service itself operates as a private core group. It contains logic to detect catalog server failures and includes quorum logic.

You need to be aware of the following failure scenarios:

- Container server failure

If the catalog service marks a container JVM as failed and the container is later reported as alive, the container JVM will be told to shut down the container servers. A JVM in this state will not be visible in **xsadmin** queries. These JVMs need to be manually restarted.

- Quorum loss from catalog failure

All members of the catalog service are expected to be online. The catalog service will only respond to container events while the catalog service has quorum. Container servers will retry requests rejected by the catalog. If you attempt to start a container during a quorum loss event, the container will not start.

If quorum is lost due to a catalog server JVM failure, manually override the quorum as quickly as possible.

If quorum is lost due to a temporary network failure, no action needs to be taken, unless the outage will last for more than a few minutes. In that case, manually override the quorum.

Note that client connectivity is allowed during quorum loss. If no container failures or connectivity issues happen during the quorum loss event, clients can still fully interact with

the container servers. However, certain clients might not have access to primary or replica copies of the data until the problem is resolved.

For a full explanation of the how container servers, catalog servers, and clients behave in the event of a failure, see the *IBM WebSphere eXtreme Scale Version 7.0 Administration Guide* at this website:

ftp://ftp.software.ibm.com/software/webserver/appserv/library/v71/xSadminguide_PDF.pdf

7.2 The placement service in WebSphere eXtreme Scale

The placement service is the brain of the WebSphere eXtreme Scale run time. It looks across the WebSphere eXtreme Scale environment and ensures that the grids meet their defined deployment policies and determines where to locate (place) shards in the available container servers. It attempts to achieve balance across the containers from the perspective of having an even distribution of shards per container across the topology (irrespective of the shard type). A shard (which belongs to a partition, which belongs to a mapSet in a grid) holds data in the grid and can be the primary, a synchronous replica, or an asynchronous replica type. As the WebSphere eXtreme Scale topology changes (for example, due to container starts or stops), the placement service is notified of the event and decides how shards will be located in the new topology.

7.2.1 Where the placement service runs

The placement service runs in the catalog service, as illustrated in Figure 7-2 on page 150. While it is capable of running in every catalog service, it is only *active* in the primary catalog service. If the primary catalog server fails, one of the remaining pool of running catalog servers becomes the new primary and the placement service will be activated there.

Any state that is generated and used by the placement service, such as the active placement plan, is held in an *embedded* system grid that is internal to WebSphere eXtreme Scale. This state is replicated throughout all running catalog servers using *synchronous* replication, ensuring that, in the case of failover, the placement service will not lose any of its previous state.

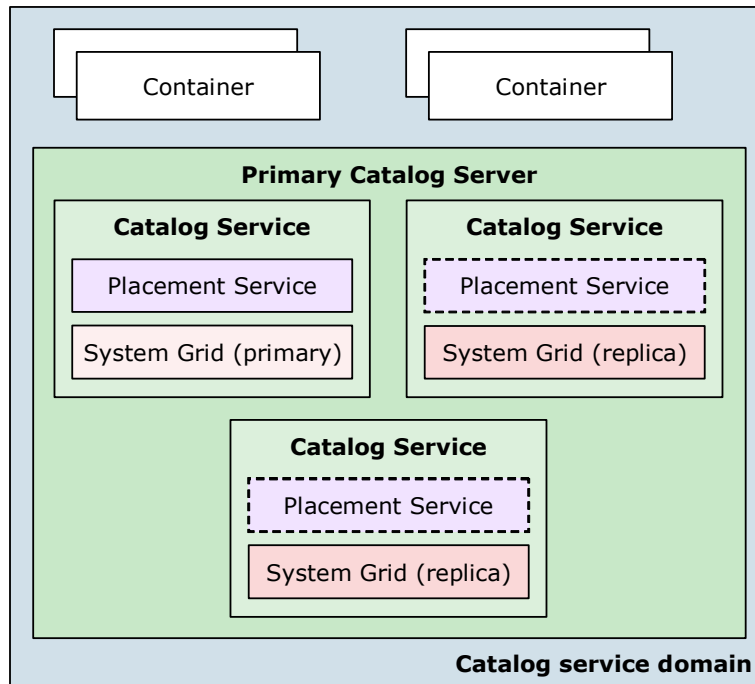


Figure 7-2 The placement service in WebSphere eXtreme Scale

To determine the current location of the primary catalog server, inspect the `SystemOut.log` file from each catalog server. The file with the most recent entry of the `CW0BJ8106I` message is the *current* primary catalog server. Example 7-6 shows an example.

Example 7-6 Determining the primary catalog server

```
CW0BJ8106I: The master catalog service cluster activated with cluster
CatalogCluster[DefaultDomain, 1 master: 2 standbys]
```

Tip: Without trace enabled, it is currently not possible to verify that a catalog server has completed synchronization with its peers. When performing a rolling restart of catalog servers, wait 5 minutes between the start of a new catalog server and stopping the primary server to allow time for the replication to complete.

7.2.2 Quorum and the placement service

The quorum mechanism protects against possible corruption of the WebSphere eXtreme Scale environment for certain topologies. When multiple catalog servers are running on a fractured network, multiple catalog servers can become the primary. As a result, there will be more than one placement service active, which leads to uncoordinated placement decisions.

When enabled, the quorum mechanism will ensure that the placement service can only run when quorum is established (or manually overridden). All catalog servers that have been defined *must* be running and able to communicate with one another over the network.

Even when quorum is enabled though, it will never be lost when you *gracefully* stop one or more catalog servers in the catalog service domain. WebSphere eXtreme Scale can detect the difference between a graceful shutdown and a failure due to the nature in which the communication sockets are closed. In the case of a graceful shutdown, the quorum will be

adjusted and the placement service will continue to be active. The following examples show the graceful stop of a catalog server:

- ▶ Stopping a stand-alone catalog server using the **stopOGServer.sh** script:

```
stopOGServer.sh <catalogServerName> -catalogServiceEndPoints  
<host1:listenerport1>,<host2:listenerport2>
```

- ▶ Stopping a catalog running in an application server, node agent, or deployment manager in an embedded environment using WebSphere Application Server systems management interfaces:
 - WebSphere administrative console
 - The **stopServer.sh** script
 - Jython script
 - JMX MBean

When to enable quorum

The quorum mechanism is disabled by default. A default heartbeat mechanism is used between the catalog servers. The detection period is approximately 30 seconds, so any short network *brownouts* (< 10 seconds) that occur do not cause placement changes to occur in the grid. Example 7-7 shows the message in the log that confirms that quorum is indeed disabled.

Example 7-7 Quorum is disabled by default

```
CW0BJ1252I: Quorum is disabled for the catalog service.
```

If your catalog service domain is contained within a single data center (or on a single local area network (LAN)), you can leave quorum disabled; however, it is a good idea to enable quorum when possible. The following examples show when to enable quorum:

- ▶ When your catalog service domain spans a network that is unpredictable or unstable (such a network might span multiple data centers)
- ▶ When you want to prevent the grid from self-healing during a brownout on the network (which will temporarily pause grid operations from occurring)

Placing catalog servers when quorum is enabled: When enabling quorum, it is *mandatory* that the catalog servers reside on separate physical machines from the container servers for failover considerations. Otherwise, if a single machine fails, quorum will not be established and the failure of the container servers cannot be mitigated until quorum is overridden.

Enabling quorum in objectGridServer.properties

Typically, quorum is enabled through the `objectGridServer.properties` file. This file is where a number of other functions, including security, can be configured. Set the **enableQuorum=true** option to enable quorum, as shown in Example 7-8.

Example 7-8 Enable quorum in objectGridServer.properties file

```
enableQuorum=true
```

For more information about the `objectGridServer.properties` file, see 5.1.3, “Server properties” on page 74.

Configuring quorum: In WebSphere Application Server, the only place to configure quorum is in the `objectGridServer.properties` file.

When quorum is enabled, you must start enough catalog servers in the domain so that quorum is established before the placement service is activated.

You need to enable quorum in the `objectGridServer.properties` file for each of the catalog servers that make up the catalog service domain. After the changes are made, restart all catalog servers. The messages that are highlighted in Example 7-9 confirm that quorum is enabled.

Example 7-9 Confirming that quorum is enabled

```
CWOB3142I: This WebSphere Application Server is not associated with a WebSphere
eXtreme Scale zone. In order to start the server in a zone, ensure that the
server's node is within a node group whose name begins with the string
ReplicationZone.
CWOB30903I: The internal version of WebSphere eXtreme Scale is v4.2.0 (7.1.0.2)
[cf21118.65559].
CWOB30913I: Server property files have been loaded:
file:/opt/IBM/WebSphere/AppServer/profiles/sa-w1201nx1-dmgr/properties/objectGridS
erver.properties.
CWOB32518I: Launching ObjectGrid catalog service: sa-w120\sa-w1201nx1-dmgr\dmgr.
CWOB31251I: Quorum is enabled for the catalog service.
CWOB32514I: Waiting for ObjectGrid server activation to complete.
CWOB31118I: ObjectGrid Server Initializing [Cluster: CatalogCluster Server:
sa-w120\sa-w1201nx1-dmgr\dmgr].
CWOB31900I: Client server remote procedure call service is initialized.
CWOB31932I: Client thread pool minimum size is 5 maximum size 50.
CWOB38401I: Waiting for a server replica to be started. Start another server(s)
immediately.
```

Enabling quorum using a command-line flag

When you are running WebSphere eXtreme Scale in a stand-alone environment, you can also enable quorum on the command-line when you start the catalog server. Example 7-10 shows a typical `startOgServer.sh` command.

Stand-alone servers only: You can use this configuration method when you start stand-alone servers only.

Example 7-10 Enabling quorum using a command-line flag

```
./startOgServer.sh cs1 -quorum true -catalogServiceEndpoints
cs1:myserver1.company.com:6601:6602,
cs2:myserver2.company.com:6601:6602,
cs3:myserver3.company.com:6601:6602
```

Manually overriding quorum

If you enable quorum, all the catalog servers must be available and communicating with the data grid to conduct placement operations. When a network brownout occurs, placement is paused until all the catalog servers are available. If a data center failure occurs, manual administrator actions are required to remove the failed catalog server from the quorum.

The `xsadmin` command (or other tool using the WebSphere eXtreme Scale JMX interface) allows an operator to override quorum so that the remaining catalog servers can continue to operate placement and other activities to keep the environment highly available. It is *critical* that before quorum is overridden, the operator ensures that the lost catalog server will not resume operations. If a network-related problem has triggered the loss of quorum, the lost catalog server must be terminated.

More information

The WebSphere eXtreme Scale Information Center has a useful section explaining catalog server quorums:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.webSphere.extremescale.admin.doc/cxsquorcatsr.html>

7.2.3 When and how the placement service runs

The placement service is only active in one of the catalog servers for the domain. The actual placement service is event driven, so during the steady state, it does nothing. The following container events are detected by the catalog service and will send an event to the placement service:

- ▶ Container start
- ▶ Container stop (teardown)
- ▶ Container process failure (JVM crash)
- ▶ Container communication loss failure (heartbeat loss)

When processing an event, there are two classes of work that the placement service generates:

- ▶ Promotions

A *promotion action* is the advancement of an existing shard to a new type: synchronous replica to primary, or asynchronous replica to synchronous replica. A promotion work item is generated as a result of failover to handle a container failure (due to stop, teardown, and so forth).

- ▶ Balances

A *balance action* is the placement of a shard to balance the distribution of shards per container (regardless of shard type). The goal of the placement computation is to immediately handle any promotion activities (that is, to deal with failures) and then to balance (evenly distribute the number of shards per container and handle the new placement of shards).

The placement service communicates with the containers that hold primary shards. Part of this communication includes the details about where the replica shards must be located. The communication to the containers, which are to host replica shards, is then performed from the primary container server.

If the primary container fails to communicate with the replica container with retries, the placement service is notified to recompute replica placement. The placement service keeps track of in-flight work and its status.

Monitoring outstanding work items: You can monitor the number of outstanding work items for the placement service by using `xsadmin.sh -placementStatus`. Refer to “`xsadmin -placementStatus`” on page 158 for more details.

How containers are selected for work items

The placement service will first generate promotion work items where host container changes are not required. Therefore, this action only includes promotions where shards are promoted.

When those work items have been generated, unassigned shards will be balanced across the available target containers. Unassigned shards can originate from donor containers or can be yet-to-be-placed due to location restrictions (for example, a replica shard will not be placed in the same container). When placing unassigned shards, the placement service uses the following order:

1. Primaries
2. Synchronous replicas
3. Asynchronous replicas

Primary and replica placement: Replicas are normally not placed into the same container as the primary. In a development environment, on a single machine, you can override this characteristic by defining the development mode attribute in the deployment policy. Assuming development mode is disabled, replicas are not placed into a container that is on the same host (IP address) as the primary shard.

The *placement strategy* that is used also influences the following behavior of the placement service:

- Placement strategy PER_CONTAINER for a grid
Primaries will always be placed on a new container as it starts
- Placement strategy FIXED_PARTITIONS
Primaries will be given placement priority until the configured number of primaries is assigned

For PER_CONTAINER placement strategy, after a replica is promoted to primary, the shard is considered to be foreign and is destroyed after all data in the shard is drained. You can obtain details about placement strategy at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extramescale.over.doc/cxsplacepart.html>

Ultimately, the placement service takes into account what has been configured in the grid's deployment configuration XML file, which directly affects the placement decisions that are made at run time.

Understanding placement during start-up

A WebSphere eXtreme Scale environment can handle a dynamic number of container servers at run time. As new containers are started, and existing containers fail or are torn down gracefully, the run time automatically adjusts the placement of shards in the grid.

There is no requirement for a fixed number of containers that will be running. Thus, when starting container servers, the placement service has to compute a new placement plan on every start event. When starting a large number of containers at the same time, this computation can cause a significant amount of work for the placement service.

To avoid this level of work, use the **numInitialContainers** configuration option in the deployment policy. This option prevents the placement service from computing or executing any placements until enough containers are started to meet or exceed the **numInitialContainers** value.

You can also manually suspend and resume the placement service balancing mechanism with the **xsadmin -suspendBalancing** and **xsadmin -resumeBalancing** commands (see 7.3.2, “Useful xsadmin commands” on page 157).

The numInitialContainers option: Setting the **numInitialContainers** option does not affect failover scenarios or any operation after the grid is up. The placement service still computes a placement plan to handle failed or stopped containers when the number of containers falls below **numInitialContainers**.

Understanding placement during shutdown

Similar to start, the placement service begins to compute a new placement plan on every container stop event. During a container stop, the placement service first puts the container into an unassigned category and computes a placement plan to accommodate all existing shards on that container as part of a *teardown* process.

When stopping a container using the **stopOGServer.sh** script, it invokes a **teardown** command, which allows the placement service to take these actions *before* the actual container process terminates. The same action applies when you use the WebSphere Application Server system management interface to stop a container server.

When stopping a number of container servers at the same time using a separate **stopOGServer** command for each container, the placement service receives a stop event for each container. However, upon receiving the first stop event, the placement service cannot know that more stop events will follow. It has to treat each stop event as an isolated event.

Use **stopOGServer** with a list of containers or use the **xsadmin -teardown** command to stop multiple container servers (see “xsadmin -teardown” on page 160). This step ensures that the placement service receives a batch of container stop events. As a result, it can compute just one new placement plan and execute this plan, thus minimizing the amount of overhead.

Using the -teardown command: When using the **xsadmin.sh -teardown** command in a WebSphere environment, the command shuts down the WebSphere eXtreme Scale containers within the WebSphere process, but it does not stop the WebSphere Application Server process. You need to follow the invocation of the **-teardown** command with a stop from normal WebSphere administrative procedures.

7.3 The xsadmin command-line tool

It is clear from the previous section that the behavior of the placement service determines what happens in the WebSphere eXtreme Scale environment. To understand what the placement service does, IBM ships a sample command-line utility called **xsadmin** that can be used for a number of functions:

- ▶ Use it after starting or stopping containers to confirm that the placement service is working as expected.
- ▶ Use it to validate that clients are able to successfully store data into the grid.
- ▶ Use it to see how many entries are in each partition. Use this information as an estimate of how much memory certain grids (or individual mapsets) are using or the distribution of memory across grids in the container if standard key/values are being used with a known size in each grid.

- Use it to look at the distribution of keys across shards. An uneven distribution of keys can indicate a poor hashing algorithm for the key that is used. Either use a separate key, or add custom code for partitioning (either in your key object's `hashCode()` method or via `PartitionableKey`).
- If the data in the grid is short-lived, use **xsadmin** to determine if there are active clients on the grid. If the total number of entries across all partitions (at the bottom of **-mapsizes** output) is non-zero, there must be active clients that put the entries there.

7.3.1 Using xsadmin in an embedded environment

The **xsadmin** command-line utility is installed for both stand-alone and embedded WebSphere eXtreme Scale environments. However, there are a number of preferred practices when using it in a (distributed) embedded environment where the catalog service and the containers are running within a WebSphere cell.

Required option: The **-dmgr** option is *required* when connecting to a catalog server running in a WebSphere Application Server process. Even when you are using a separate host and port number, you still need to include this flag.

Catalog service running in deployment manager process

Assuming that the catalog service is running in the deployment manager and has been set up using default ports, use the following syntax when running **xsadmin** from the deployment manager host. The utility will attempt to connect to the catalog service on the local host:

```
xsadmin.sh -containers -dmgr
```

Catalog service running in multiple WebSphere processes

Example 7-11 shows how **xsadmin.sh** is run against a catalog service domain running across three separate WebSphere processes. The option **-cep** is used to specify the host and port combinations for the three catalog servers in the domain. The port that is used here is the bootstrap address of the process.

Example 7-11 Using the -dmgr flag

```
[wasuser@sa-w1201nx1 bin]$ ./xsadmin.sh -dmgr -cep  
sa-w1201nx1:9809,sa-w1201nx2:2809,sa-w1201nx3:2809 -containers
```

Catalog service running with WebSphere security enabled

Most WebSphere Application Server environments will have WebSphere security enabled. In these environments, the **xsadmin** utility will work fine, but it will prompt for authentication upon start-up. Because this command is typically run a number of times, configure authentication through a properties file.

For authentication, the **xsadmin** utility relies on the following Secure Association Service (SAS) client properties file:

```
$WAS_PROFILE_HOME/properties/sas.client.props
```

Example 7-12 on page 157 shows what needs to be changed in order to use authentication with the user name and password provided in the properties file.

Example 7-12 Changes made to sas.client.props

```
# com.ibm.CORBA.loginSource=stdin
com.ibm.CORBA.loginSource=properties

# RMI/IIOP user identity
com.ibm.CORBA.loginUserId=username
com.ibm.CORBA.loginPassword=password
```

7.3.2 Useful xsadmin commands

Several options can be passed into the command-line utility to extract various pieces of information from the placement service. The commands that are provided here are a subset of what is available from the tool. We think that these commands are the most commonly used commands.

xsadmin -mapsizes

You can wrap the invocation of the **xsadmin -mapsizes** command with a script that invokes it and parses its output into a custom piece of monitoring data. You can invoke the command as a cron job and compare the result against a configured threshold value to trigger a custom alert. Example 7-13 shows an example of the output from the **xsadmin -mapsizes** command. The command is **xsadmin.sh -mapsizes**.

Example 7-13 Sample xsadmin -mapsizes output

```
[wasuser@sa-w120lnx1 bin]$ ./xsadmin.sh -mapsizes -dmgr
```

This Administrative Utility is provided as a sample only and is not to be considered a fully supported component of the WebSphere eXtreme Scale product

Connecting to Catalog service at localhost:9809

```
*****Displaying Results for Grid - UserGrid, MapSet -
UserMapSet*****
```

```
*** Listing Maps for sa-w120\sa-w120lnx2\sa-w120lnx2_container1 ***
```

Map Name	Partition	Map Size	Used Bytes (B)	Shard Type
User	2	0	0	SynchronousReplica
User	4	0	0	Primary
User	5	0	0	SynchronousReplica
User	6	0	0	Primary
User	9	0	0	SynchronousReplica
User	10	0	0	Primary
User	12	0	0	Primary
User	15	0	0	Primary

```
Server Total: 0 (0B)
```

```
Total Domain Count: 0 (0B)
```

```
(The used bytes statistics are accurate only when you are using simple objects or
the COPY_TO_BYTES copy mode.)
```

If you run the **xsadmin -mapsizes** command and placement has not yet happened (for example, the number of started containers is less than **numInitialContainers**), the output from the command will not list any containers or partitions. It will only include the “Total Domain Count:” line.

xsadmin -containers

The **xsadmin -containers** command (Example 7-14) will display the placement service's view of the topology from its current computed placement strategy. The output can be used during starts and stops and general checks to ensure that the appropriate number of containers is running in the topology and to verify the distribution of the shards.

Example 7-14 Sample xsadmin -containers output

```
[wasuser@sa-w120lnx1 bin]$ ./xsadmin.sh -containers -dmgr
```

This Administrative Utility is provided as a sample only and is not to be considered a fully supported component of the WebSphere eXtreme Scale product

Connecting to Catalog service at localhost:9809

*** Show all online containers for grid - UserGrid & mapset - UserMapSet

Host: sa-w120lnx3.itso.ral.ibm.com

Container: sa-w120\sa-w120lnx3\sa-w120lnx3_container1_C-0,
Server:sa-w120\sa-w120lnx3\sa-w120lnx3_container1, Zone:DefaultZone

Partition	Shard Type	Reserved
3	Primary	false
5	Primary	false
9	Primary	false
13	Primary	false
14	Primary	false
1	SynchronousReplica	false
11	SynchronousReplica	false
15	SynchronousReplica	false

xsadmin -placementStatus

The **xsadmin -placement** status command (Example 7-15) shows counters for known topology artifacts, such as the number of containers and the number of hosts, as well as a counter for any outstanding work that the placement service currently has. During the steady state, there must not be any outstanding work items for the placement service. Additionally, the container and host count need to match what is expected for the known topology. A cron job can be created to invoke this command at regular intervals and parse the output against known numbers, which can provide a mechanism for being alerted to failed or down containers or hosts.

The **xsadmin** sample utility provides a command for displaying the placement status for each mapset in each grid. You can use the **-g grid_name** and **-m mapset_name** filter parameters to narrow the results. You can monitor the *numOutstandingWorkItems* to ensure that it drains to 0.

Example 7-15 Sample xsadmin placementStatus output

```
[wasuser@sa-w120lnx1 bin]$ ./xsadmin.sh -placementStatus -dmgr
```

This Administrative Utility is provided as a sample only and is not to be considered a fully supported component of the WebSphere eXtreme Scale product

Connecting to Catalog service at localhost:9809

*****Printing Placement Status for Grid - UserGrid, MapSet -
UserMapSet*****

```

<objectGrid name="UserGrid" mapSetName="UserMapSet">
  <configuration>
    <attribute name="placementStrategy" value="FIXED_PARTITIONS"/>
    <attribute name="numInitialContainers" value="4"/>
    <attribute name="minSyncReplicas" value="0"/>
    <attribute name="developmentMode" value="false"/>
  </configuration>
  <runtime>
    <attribute name="numContainers" value="4"/>
    <attribute name="numMachines" value="2"/>
    <attribute name="numOutstandingWorkItems" value="0"/>
  </runtime>
</objectGrid>

```

xsadmin -unassigned

The **xsadmin -unassigned** command (Example 7-16) displays any shards that are currently not placed on target containers. During the steady state, there must not be any unassigned shards. You can create a cron job to invoke this command at regular intervals and parse the output to ensure that no unassigned partitions exist. Unassigned partitions can be an indication of a down container or host. The command is **xsadmin -unassigned**.

Example 7-16 Sample xsadmin -unassigned output

```
[wasuser@sa-w1201nx1 bin]$ ./xsadmin.sh -unassigned -dmgr
```

This Administrative Utility is provided as a sample only and is not to be considered a fully supported component of the WebSphere eXtreme Scale product

```

Connecting to Catalog service at localhost:9809
*** Showing unassigned containers for grid - UserGrid & mapset - UserMapSet
Partition Shard Type

```

xsadmin -routetable

The **xsadmin -routetable** command (Example 7-17) simulates a new client connection to the data grid and displays each shard state (reachable, unreachable, and invalid) for the grids. During normal steady state operation, all shards must be reachable. You can create a cron job to invoke this command and parse the output for unreachable or invalid states. The command is **xsadmin.sh -routetable**.

Example 7-17 Sample route table with shards in an unhealthy state

```
[wasuser@sa-w1201nx1 bin]$ ./xsadmin.sh -routetable -dmgr
```

This Administrative Utility is provided as a sample only and is not to be considered a fully supported component of the WebSphere eXtreme Scale product

```
Connecting to Catalog service at localhost:9809
```

```
*****Displaying Routing Info for Grid - UserGrid*****
```

Shard Type	Partition	State	Host	Zone
Primary	0	reachable	sa-w1201nx3.itso.ra1.ibm.com	DefaultZone
Replica	0	reachable	sa-w1201nx2.itso.ra1.ibm.com	DefaultZone
Replica	0	reachable	sa-w1201nx2.itso.ra1.ibm.com	DefaultZone

Primary	1	reachable	sa-w1201nx2.itso.ra1.ibm.com	DefaultZone
Replica	1	reachable	sa-w1201nx3.itso.ra1.ibm.com	DefaultZone
Primary	2	reachable	sa-w1201nx3.itso.ra1.ibm.com	DefaultZone
Replica	2	reachable	sa-w1201nx2.itso.ra1.ibm.com	DefaultZone
...				
Replica	14	reachable	sa-w1201nx2.itso.ra1.ibm.com	DefaultZone
Primary	15	reachable	sa-w1201nx2.itso.ra1.ibm.com	DefaultZone
Replica	15	reachable	sa-w1201nx3.itso.ra1.ibm.com	DefaultZone

If you learn that there are shards that are in a consistently unreachable or unknown/invalid state, placement can be forced to rerun a placement plan that might clear up the issue of unhealthy shards. Placement can be forced to run again using a invocation of the **xsadmin -triggerPlacement** command.

Security: The **xsadmin -routetable** command might produce an “unreachable” error if you enable security. An open authorized program analysis report (APAR) PM43412 exists to track this issue:

<http://www-01.ibm.com/support/docview.wss?uid=swg1PM43412>

xsadmin -quorumStatus

The **xsadmin -quorumStatus** command (Example 7-18) displays the current status of quorum among the catalog servers. Use this command if you think you have lost quorum. The command is **xsadmin.sh -quorumStatus**.

Example 7-18 Sample xsadmin -quorumStatus output

```
[wasuser@sa-w1201nx1 bin]$ ./xsadmin.sh -unassigned -dmgr
```

This Administrative Utility is provided as a sample only and is not to be considered a fully supported component of the WebSphere eXtreme Scale product

Connecting to Catalog service at localhost:9809

```
Active Servers - 3
Quorum Requirement - 3
```

xsadmin -teardown

The **xsadmin -teardown** command (Example 7-19) is used to stop a list or group of catalog and container servers. This command simplifies shutting down all or portions of a data grid, avoiding unnecessary placement and recovery actions by the catalog service that normally occur when processes are stopped or killed. The format is **xsadmin.sh -teardown server_name [, server_name]**.

Example 7-19 Sample xsadmin -teardown output

```
[wasuser@sa-w1201nx1 bin]$ ./xsadmin.sh -dmgr -teardown
sa-w1201\sa-w1201nx2\sa-w1201nx2_container1,sa-w1201\sa-w1201nx3\sa-w1201nx3_co
ntainer1
```

This Administrative Utility is provided as a sample only and is not to be considered a fully supported component of the WebSphere eXtreme Scale product

Connecting to Catalog service at localhost:9809

The following servers will be torn down:

```
sa-w120\sa-w120lnx2\sa-w120lnx2_container1
sa-w120\sa-w120lnx3\sa-w120lnx3_container1
```

Do you want to tear down the listed servers? (Y/N)

Y

Teardown results:

```
Server sa-w120\sa-w120lnx2\sa-w120lnx2_container1 torn down successfully -
true
Server sa-w120\sa-w120lnx3\sa-w120lnx3_container1 torn down successfully -
true
```

xsadmin -suspendBalancing and -resumeBalancing

Command availability: The commands that are described here are currently only available after applying APAR PM37461 on top of WebSphere eXtreme Scale 7.1.0.2 (Fix Pack 2). This APAR is publicly available and can be downloaded from this website:

<http://www-01.ibm.com/support/docview.wss?uid=swg24029863>

The **xsadmin -suspendBalancing** command (Example 7-20) allows an administrator to disable the balancing mechanism of the placement service. No balance work is computed or executed by the placement service. However, placement work continues to occur to ensure that failover of shards is not affected.

The command is **xsadmin.sh -suspendBalancing *objectGrid_Name mapSet_Name***.

Example 7-20 Sample xsadmin -suspendBalancing output

```
[wasuser@sa-w120lnx1 bin]$ ./xsadmin.sh -suspendBalancing UserGrid UserMapSet
-dmgr
```

This Administrative Utility is provided as a sample only and is not to be considered a fully supported component of the WebSphere eXtreme Scale product

Connecting to Catalog service at localhost:9809

```
*****Printing Suspend Balancing Results for Grid - UserGrid, MapSet -
UserMapSet*****
```

```
<objectGrid name="UserGrid" mapSetName="UserMapSet">
  <suspendBalancing currentValue="true" previousValue="false"/>
</objectGrid>
```

You can use the **xsadmin -resumeBalancing** command to enable the balancing mechanism again.

xsadmin -triggerPlacement

The **xsadmin -triggerPlacement** command simulates a new client connection to the data grid and displays each shard state (reachable, unreachable, and invalid) for the grids. During normal steady state operation, all shards must be reachable. You can create a cron job to invoke this command and parse the output for unreachable or invalid states.

The command is `xsadmin.sh -triggerPlacement grid_name mapSet_Name`.

If entries in the routetable remain unreachable: If `xsadmin -triggerPlacement` does not work (that is routetable entries are still in an unknown or unreachable state), capture the thread dumps of the container that hosts the shard, and restart the container to force the routetable to be cleaned up. Preserve the thread dumps and all environment logs for problem analysis.

7.4 Configuring failure detection

WebSphere eXtreme Scale provides high availability and fault tolerance by design. However, it is important to understand how failure detection works. Several of the default time-outs that are used by the product are rather generous, so we recommend that you review those time-outs. This section provides an overview of the components that have an impact on the speed at which failures are detected.

Figure 7-3 on page 163 shows the components and configuration options that are available at the client and container (server).

Testing is key: The configuration options that are presented here must always be reviewed in the context of the application's workflow, the topology and network infrastructure, and the requirements for failover. We strongly recommend to test before implementing these configuration options in a production environment. Ultimately, the goal is to find a balance between the fast detection of failures and a minimum of false positives.

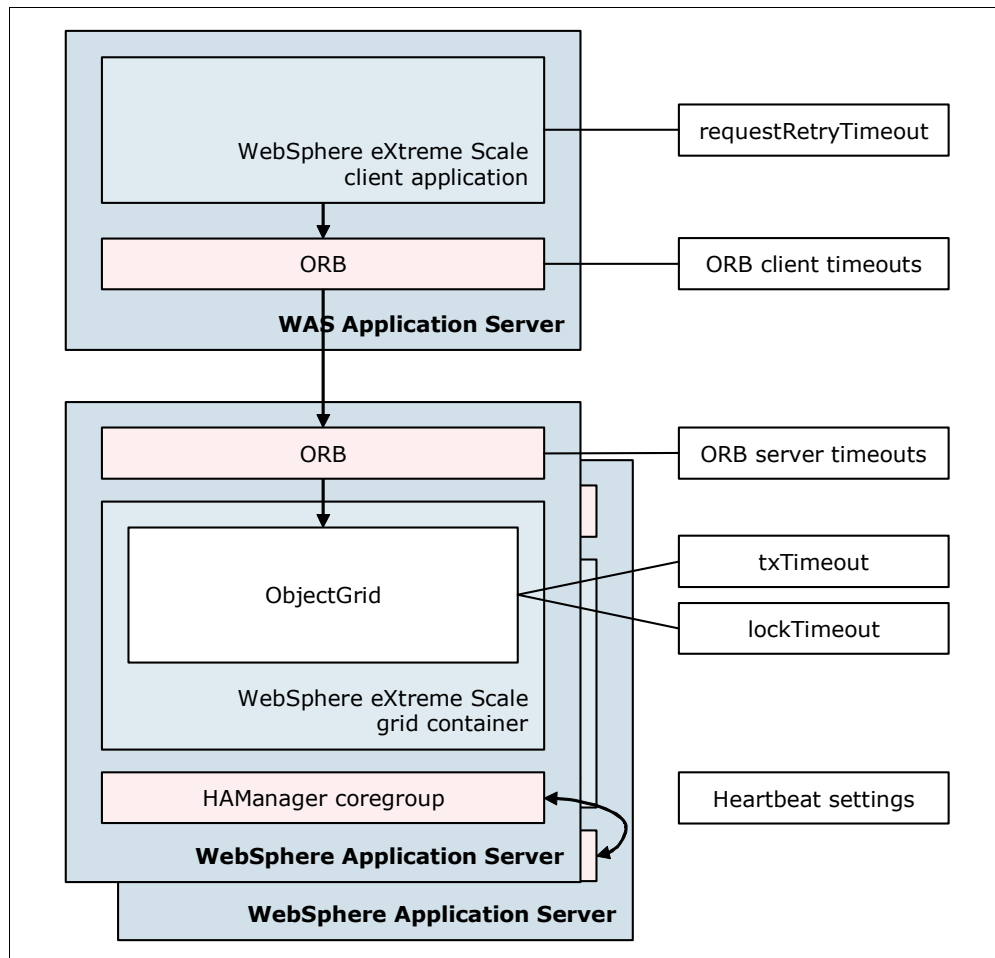


Figure 7-3 Components that have an impact on the speed at which failures are detected

Note that you also need to tune the operating system for efficient operation and failure detection. The following page in the information center provides a good starting point. The tuning is focused around network I/O.

<http://publib.boulder.ibm.com/infocenter/wxinfo/v7r1/index.jsp?topic=/com.ibm.webSphere.extremescale.admin.doc/cxsopernetw.html>

7.4.1 Container failover detection

This section describes settings that can be used to configure the failover detection on the containers.

Setting container ORB time-outs

The settings in Example 7-21 show how to set the failover detection for the ORB to 5 seconds. ORB properties can be set with an `orb.properties` file, as application server settings in the administrative console, or as custom properties on the ORB in the administrative console.

Example 7-21 orb.properties configuration file

```
com.ibm.CORBA.RequestTimeout=5
com.ibm.CORBA.ConnectTimeout=5
```

```
com.ibm.CORBA.FragmentTimeout=5
com.ibm.CORBA.LocateRequestTimeout=5
```

Setting core group heartbeat intervals

WebSphere eXtreme Scale relies on a heartbeat mechanism to detect whether containers are up and running. The actual implementation is based on the core groups that are used by WebSphere's High Availability Manager.

Failures because of process crashes (or *soft* failures) are typically detected in less than one second. The network sockets are automatically closed by the operating system hosting the process when a soft failure occurs. A *hard* failure is a physical computer or server crash, network cable disconnection, or operating system error. In this case, it can take much longer for WebSphere eXtreme Scale to detect that a container has failed. To ensure that hard failures are detected within a reasonable amount of time, configure the heartbeat mechanism.

Heartbeat interval timing: An aggressive heartbeat interval can be useful when the processes and network are stable. If the network or processes are not optimally configured, heartbeats might be missed, which can result in a false failure detection.

Configuring failover for stand-alone environments

You can configure heartbeat intervals on the command line by using the **-heartbeat** parameter of the **startOgServer.sh** command-line script. Table 7-1 shows the available options.

Table 7-1 Heartbeat interval options for stand-alone environments

Value	Failover detection	Description
0	Typical (default)	Failovers are typically detected within 30 seconds.
-1	Aggressive	Failovers are typically detected within 5 seconds.
1	Relaxed	Failovers are typically detected within 180 seconds.

Configuring failover for WebSphere Application Server V6 environments

You can configure WebSphere Application Server Network Deployment 6.0.2 and 6.1 to control the WebSphere eXtreme Scale failover behavior. The default failover time for hard failures is approximately 200 seconds, which is relatively long.

WebSphere eXtreme Scale running in a WebSphere Application Server process inherits the failover characteristics from the core group settings of the application server. Table 7-2 on page 165 lists the custom properties that are used to configure the core group heartbeat settings for various versions of WebSphere Application Server Network Deployment V6 and V6.1. These properties are specified using custom properties on the core group using the WebSphere administrative console and must be specified for all core groups that are used by the application.

You must also specify the number of missed heartbeats with the **IBM_CS_FD_CONSECUTIVE_MISSED** property. The value indicates how many heartbeats can be missed before a peer JVM is considered as failed. The hard failure detection time is approximately the product of the heartbeat interval and the number of missed heartbeats.

Table 7-2 Core group custom properties to control heartbeat settings

Version	Custom properties	Description	Default value
6.0.2.0 and higher	IBM_CS_FD_PERIOD_SEC	Heartbeat interval in seconds	20
6.1.0.0 - 6.1.0.12	IBM_CS_FD_PERIOD_SEC	Heartbeat interval in seconds	20
6.1.0.13 and higher	IBM_CS_FD_PERIOD_MILLIS	Heartbeat interval in milliseconds (ms)	None
6.x	IBM_CS_FD_CONSECUTIVE_MISSED	Number of missed heartbeats required	10

So, as an example, you configure the following core group custom properties to achieve a 1500 ms failure detection time for WebSphere Application Server Network Deployment 6.1.0.13 or higher servers:

- ▶ Set IBM_CS_FD_PERIOD_MILLIS = 750
- ▶ Set IBM_CS_FD_CONSECUTIVE_MISSED = 2

Configuring failover for WebSphere Application Server 7.0 environments

The following core group settings control the failover detection behavior for WebSphere Application Server Network Deployment 7.0:

- ▶ Heartbeat transmission period (default is 30000 milliseconds)
- ▶ Heartbeat timeout period (default is 180000 milliseconds)

Important: These settings affect the entire WebSphere Application Server cell, but they have no effect on clients.

To find and change these settings using the WebSphere administrative console:

1. Go to **Servers** → **Core Groups** → **Core group settings**.
2. Select the appropriate core group (for example, **DefaultCoreGroup**).
3. Go to **Discovery and failure detection**.
4. Change the *Heartbeat transmission period* and *Heartbeat timeout period* to suit your needs.

Figure 7-4 on page 166 shows how to configure these settings to achieve a 1500 ms failure detection time.

Figure 7-4 Configuring heartbeat timeouts in WebSphere Network Deployment 7.0 environments

7.4.2 Client failure detection

This section describes settings that can be used to configure failure detection on the clients. In addition to the settings that are described here, you can also set the ORB ("Setting container ORB time-outs" on page 163).

Setting BackingMap time-outs on the container

Example 7-22 shows settings in the `objectGrid.xml` that configure a 3 second time-out detection of a client.

Example 7-22 Setting time-outs on BackingMap in the `objectGrid.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="UserGrid" txTimeout="3" lockTimeout="3">
      <backingMap name="User"/>
    </objectGrid>
  </objectGrids>
</objectGridConfig>
```

lockTimeout

The **`lockTimeout`** setting configures the time-out that is used by the lock manager for the BackingMap instance. Set the `lockStrategy` attribute to `OPTIMISTIC` or `PESSIMISTIC` to create a lock manager for the BackingMap instance. To prevent deadlocks from occurring, the lock manager has a default time-out value of 15 seconds. If the time-out limit is exceeded, a `LockTimeoutException` occurs. The default value of 15 seconds is sufficient for most

applications, but on a heavily loaded system, a time-out might occur when no deadlock exists. Use the `lockTimeout` attribute to increase the value from the default to prevent false time-out exceptions from occurring. Set the `lockStrategy` attribute to `NONE` to specify that the `BackingMap` instance use no lock manager (optional).

For more information about locking strategy, see 6.3, “Locking performance preferred practices” on page 122.

txTimeout

The **`txTimeout`** setting specifies the amount of time in seconds that a transaction is allowed for completion. If a transaction does not complete in this amount of time, the transaction is marked for rollback and a `TransactionTimeoutException` results. If you set the value to 0, transactions never time out.

Setting requestRetryTimeout on the client

Clients can be configured to retry transactions after a time-out. The time-out value is specified with the `requestRetryTimeout` property on the client, which is specified in the client properties file (`objectGridClient.properties`, by default). If set to a value greater than 0, the request will be retried on exceptions for which retry is available. Set the value to 0 to fail without retries on exceptions. The retry logic will run until the **`requestRetryTimeout`** (in ms) is reached.

The client will retry a transaction again if it encounters one of the following exceptions:

- ▶ `ReplicationVotedToRollbackTransactionException` (only on autocommit)
- ▶ `TargetNotAvailable`
- ▶ `org.omg.CORBA.SystemException`
- ▶ `AvailabilityException` (only on autocommit)
- ▶ `LockTimeoutException` (only on autocommit)
- ▶ `UnavailableServiceException` (only on autocommit)

The client will *not* retry transactions when it encounters one of the following *permanent* exceptions:

- ▶ `DuplicateKeyException`
- ▶ `KeyNotFoundException`
- ▶ `LoaderException`
- ▶ `TransactionAffinityException`
- ▶ `LockDeadlockException`
- ▶ `OptimisticCollisionException`

For more information, see this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extramescale.admin.doc/txsreblmaps.html>

7.5 Monitoring WebSphere eXtreme Scale

There are a number of tools and techniques to determine what is happening in a WebSphere eXtreme Scale environment during run time. In this section, we provide an insight in those tools and techniques and help you choose which tools and techniques to implement for monitoring a production environment. We explain the following tools and techniques in this section:

- ▶ “Operating system monitoring” on page 168
- ▶ “Monitoring WebSphere eXtreme Scale logs” on page 169
- ▶ “WebSphere eXtreme Scale web console” on page 171

- ▶ “Monitoring with Tivoli Performance Viewer” on page 178
- ▶ “Monitoring using a WebSphere eXtreme Scale ping client” on page 180
- ▶ “Additional monitoring tools” on page 181

7.5.1 Operating system monitoring

It is common to monitor the operating system for most enterprise middleware, including WebSphere eXtreme Scale. Typically, the operating system processes and resource utilization are captured over a period of time, which can be used as input into capacity planning.

Processes

A WebSphere eXtreme Scale environment generally consists of a number of Java components that are deployed across a number of JVMs. It is recommended that you monitor whether those JVM processes are running.

Additional monitoring: Note that a running process does not guarantee the health of the environment. Additional monitoring is recommended.

CPU resources

The catalog and container server processes need to have sufficient CPU resources at their disposal to maintain a healthy system. If a process is suffering from *CPU starvation*, it can lead to communication breakdown and cause instability within the WebSphere eXtreme Scale environment.

CPU starvation is detected by the JVM and the standard JVM “CPU Starvation Detected” message will appear in the `SystemOut.log`. Note that the CPU starvation message is also typically seen when the real problem is running out of memory. CPU time spent in garbage collection (GC) increases as you get near an out-of-memory condition, and this same error message is seen prior to, or often instead of, an actual `OutOfMemoryError` message.

You must monitor CPU resources on all systems to ensure that the client, catalog, and container server processes can obtain sufficient CPU cycles to perform the necessary work in time. Monitoring tools, such as IBM Tivoli Composite Application Manager, or third-party tools, such as HP Diagnostics Software, can provide insight into operating system and process CPU utilization. Alternatively, operating system utilities, such as `top`, `vmstat`, or `nmon`, can be used to gather this information.

Investigate unusually high CPU consumption of WebSphere eXtreme scale processes, which might be indicative of suboptimal garbage collection, runaway threads, or another issue. Before restarting the process, you need to collect JVM thread dumps for evaluation and a potential solution later.

Memory resources and paging

Modern operating systems move memory pages to disk if there is insufficient available physical memory. WebSphere eXtreme Scale (like any Java application) does not tolerate paging, so avoid this action. One possible exception is z/OS® where tests have shown that you can overcommit up to 30% of memory with less than a 10% reduction in JVM performance. This trade-off might be worthwhile for certain situations. Note that, as of WebSphere eXtreme Scale V7.1, only clients (not catalogs or containers) can run under z/OS; Linux for System z is supported for clients, catalogs, and containers.

If paging occurs, deal with it immediately. Most operating system utilities and third-party monitoring tools can monitor paging.

The network

For a distributed grid, network bandwidth is another key resource. WebSphere eXtreme Scale will multiplex over a single ORB connection all work, for all threads in a given client JVM, that pertains to a given container JVM. One client accessing data over 16 containers will have 16 connections, no matter how many threads or partitions are used. The available bandwidth in your network must be able to handle the required traffic for your target performance (response time and throughput at load). The ORB (and thus WebSphere eXtreme Scale) can make use of multiple network interface cards (NICs) for a single JVM, which can improve performance. Monitoring network performance is a key aspect of monitoring a distributed grid. Because you can configure a host (IP address) along with ports for catalogs and containers, you can also cause certain catalogs and containers to use other NICs by assigning each NIC its own IP address.

7.5.2 Monitoring WebSphere eXtreme Scale logs

WebSphere eXtreme Scale has been instrumented to log messages and generate first-failure data capture (FFDC) reports when runtime problems occur. Although, under steady state conditions, you typically do not see a lot of messages being logged.

However, not all runtime problems necessarily produce error messages in the `SystemOut.log`. It is therefore *vital* to monitor the FFDC logs, as well. For example, you can set up a cron job to periodically perform a `grep` across the logs for error messages. Alternatively, you can set up tooling, such as IBM Tivoli Monitoring log scraping agents, to look for particular message patterns and raise alerts when they occur.

In this section, we list a number of common error messages and error conditions for WebSphere eXtreme Scale environments. Use these messages as a starting point when setting up one of the previous mechanisms to monitor the logs and FFDC data.

Transaction time-outs

There are a variety of log messages for various time-out errors, such as `TransactionTimeout`, `request timeout`, `connectionexception`, and so forth. The following list shows possible exceptions or text that can indicate various error conditions:

- ▶ `ConnectException` while trying to call `ObjectGridManager.connect()`
- ▶ `ObjectGridException` while calling `ObjectGridManager.getObjectGrid()`, `ObjectGrid.getSession()`, `Session.getMap()`, `ObjectMap.get()`, `put()`, and so forth
- ▶ `TargetNotAvailable`
- ▶ `org.omg.CORBA.SystemException`
- ▶ `AvailabilityException` (only on autocommit)
- ▶ `UnavailableServiceException` (only on autocommit)
- ▶ `TransactionTimeoutException`

LockTimeoutException

Both catalog servers and container servers can log this exception to the `SystemOut.log`. However, the recommended actions differ:

- Catalog server

If the exception occurs on a *catalog server*, restart the catalog server as soon as possible. Be careful when quorum is enabled for the catalog servers.

- Container server

If the exception occurs on a *container server*, it is a separate matter. Depending on the locking strategy, WebSphere eXtreme Scale uses locks to ensure the consistency of data in the `BackingMap`. When you observe `LockTimeoutExceptions` on the container servers, the locking strategy is unsuitable for the application and workload. Resolving this problem most likely requires changes to the application or the locking strategy. Refer to 6.3, “Locking performance preferred practices” on page 122 for more details about the locking strategy.

java.lang.Error: Maximum lock count exceeded

This exception can appear in the FFDC logs. When encountered, restart the container server immediately.

CWOBJ1123W

This message in `SystemOut.log` indicates that the catalog or container server has become detached from the grid. Example 7-23 shows the message.

When this message is logged, the server no longer participates in any workload. Restart the server as soon as possible. Sometimes, it might be necessary to issue a `kill` command against the process ID to stop the server.

Example 7-23 CWOBJ1123W indicating that a server has become detached from the grid

```
[2/24/11 7:01:43:747 CST] 7ea27ea2 ServerAgent W CWOBJ1123W: Server was  
disconnected from the primary catalog service and cannot be reconnected.
```

ClassNotFoundException

A `ClassNotFoundException` in the `SystemOut.log` typically indicates an application problem or a mismatch of the application code that is deployed on the various WebSphere eXtreme Scale servers or clients. The same situation applies to a `ClassCastException` and a `NoClassDefFoundException`.

These messages certainly warrant an overall verification of the versions of the various application components that are deployed. A new deployment might be required to resolve this situation.

OutOfMemoryException

An `OutOfMemoryException` in the `SystemOut.log` typically means that the JVM is unable to allocate memory for an object within the JVM heap. The size of the heap is insufficient to accommodate all the objects of the Java application. In most cases, the JVM process cannot recover from this situation and has to be restarted.

Although memory leaks can be the cause of this situation, in a WebSphere eXtreme Scale environment, consider the capacity of the grid first. For example, when you run a grid that might contain up to 10 GB of data on four JVMs with a maximum heap size of 2 GB each,

your capacity is insufficient and you risk `OutOfMemoryExceptions`. See 4.1, “Planning for capacity” on page 50 for more information about capacity planning.

NotSerializableException

WebSphere eXtreme Scale relies on standard Java serialization and deserialization to send objects from one process to another. When a `NotSerializableException` is logged in the `SystemOut.log`, it usually means that either the key or, more likely, the value object of a key/value pair cannot be serialized.

You can avoid this exception by either implementing the `Externalizable` or `Serializable` interface or by using an `ObjectTransformer` plug-in. See 6.4, “Serialization performance” on page 127 for more information.

Replication-related exceptions

The following messages in the `SystemOut.log` of a container server indicate replication-related problems:

- ▶ CWOBJ1524I
A problem occurred during replication and a recovery attempt will be made. This message is an informational message. If the recovery is successful, this message is benign.
- ▶ CWOBJ1537E
The replica has made an attempt to recover using retry logic; however, it has failed and is now *offline*. Use `xsadmin` commands to determine whether the replica was replaced or whether it is still unassigned. You might need to restart the container server.
- ▶ CWOBJ1527W
The replica is now *offline*. A fatal system error occurred when it made an attempt to replicate from the primary. Use `xsadmin` commands to determine whether the replica was replaced or is still unassigned. You might need to restart the container server.
- ▶ CWOBJ1533E
There was a *state transition problem* with a replica, leaving it in an unknown state. Data might have been corrupted in the process, and you need to restart the container server immediately.

FFDC incidents

As mentioned earlier, a number of WebSphere eXtreme Scale runtime exceptions do not log anything in the `SystemOut.log`. FFDC incidents are opened when those exceptions are encountered, so you need to monitor the FFDC incidents on a regular basis.

Look at the entries in the `ffdc_exception.log` to determine the number of WebSphere eXtreme Scale FFDC incidents from start-up. Large counts that are associated with WebSphere eXtreme Scale runtime packages warrant a closer look:

- ▶ `com.ibm.ws.objectgrid`
- ▶ `com.ibm.websphere.objectgrid`

7.5.3 WebSphere eXtreme Scale web console

The WebSphere eXtreme Scale web console is a new feature with Version 7.1. It is primarily used for sizing and performance-related information. However, it can be useful in a production environment as well.

The web console is only installed with the stand-alone version of WebSphere eXtreme Scale. So, even when you use WebSphere eXtreme Scale in a WebSphere Application Server environment, you still need to install the stand-alone product.

You can install the stand-alone product on the same machine as, for example, the deployment manager. But, you can also install it on a separate machine. However, remember that the console server uses Remote Method Invocation (RMI)/Internet Inter-ORB Protocol (IIOP) and needs to connect to the ORB listener port of the catalog servers.

Network firewall note: Avoid placing a network firewall between the system that runs the console and the systems that run the catalog service. The RMI/IIOP protocol cannot be handled by certain firewalls, which can result in connectivity problems.

Supported platforms for running the console

The following platforms are supported for running the console:

- ▶ AIX
- ▶ Linux
- ▶ Windows

Solaris and HP/UX: Note that there is no support for Solaris or HP/UX at the time of writing this book. A *work-around* for the Solaris platform is available though, as described in “Special installation instructions for 64-bit Windows operating systems” on page 172.

Web browser requirements

The web console *user interface* supports the following web browsers:

- ▶ Mozilla Firefox, Version 3.5.x and later
- ▶ Mozilla Firefox, Version 3.6.x and later
- ▶ Microsoft Internet Explorer, Version 7 or 8

Special installation instructions for 64-bit Windows operating systems

Although WebSphere eXtreme Scale is supported on 64-bit Microsoft Windows operating systems, the web console is not supported.

Windows operating system

When you install WebSphere eXtreme Scale stand-alone on a 64-bit Microsoft Windows operating system and attempt to run the **startConsoleServer.bat** command to start to console, it will fail with the following error:

```
CWPZC8029E: ZS0 could not be located for the current platform of OS=windows, arch=x86_64.
```

This error is also documented at this website:

<http://www-01.ibm.com/support/docview.wss?rs=3023&context=SSPPLQ&q1=v71xsrnotes&uid=swg21438057>

However, there is a work-around. When you run the **install.bat** command to install WebSphere eXtreme Scale on a 64-bit Microsoft Windows operating system, it automatically launches the 64-bit installer (which will install the 64-bit version of WebSphere eXtreme Scale).

You can also install the 32-bit version of WebSphere eXtreme Scale on a 64-bit Microsoft Windows operating system. Instead of running the `install.bat` command, launch the 32-bit installer directly with the following command:

```
WXS/install.windows.ia32.exe
```

AIX operating system

If you attempt to run `startConsoleServer` on an AIX system, you will see an error similar to the following error that is shown in Example 7-24.

Example 7-24 Error resulting from attempting to run startConsoleServer on an AIX system

```
wxs@wxshost01:/usr/IBM/WebSphere/eXtremeScale/ObjectGrid/bin>
./startConsoleServer.sh
[error]The WebSphere eXtreme Scale console server is only supported with
32-bit Java Runtime Environments (JRE). Set the JAVA_HOME environmental variable
prior to starting the console server to reference a supported 32-bit JRE, or
install the 32-bit version of WebSphere eXtreme Scale.
```

The work-around is simple:

1. Use another 32-bit JVM by updating `JAVA_HOME` (which is located in the script itself).
2. Remove the `resolved.properties` file at
`$OBJECTGRID_HOME/console/.zero/private/resolved.properties`.
Removing this file clears the cache that the underlying project zero engine resolves for the native path first.

If you forget to remove the `resolved.properties` file, you get message CWPZC0011E with `UnsatisfiedLinkError` being thrown (Example 7-25) when you attempt to start the console.

Example 7-25 Error resulting from not clearing the resolved.properties file

```
wxs@wxshost01:/usr/IBM/WebSphere/eXtremeScale/ObjectGrid/bin>
./startConsoleServer.sh

CWPZC0011E: An unhandled error was caught; message:
java.lang.UnsatisfiedLinkError: ZeroProcessManagement (A file or directory in the
path name does not exist.)
```

After you have completed these two steps, you can successfully start the web console (Example 7-26).

Example 7-26 Successful start of the web console

```
wxs@wxshost01:/usr/IBM/WebSphere/eXtremeScale/ObjectGrid/bin>
./startConsoleServer.sh

CWPZT0600I: Command resolve was successful
[info]Creating new XS database using schema version: 1.0.0.2
[info]Parsing file ./sql/derby/create-1.0.0.2.sql...
[info]Finished executing file ./sql/derby/create-1.0.0.2.sql.
[info]Creating acl tables
[info]Parsing file
/usr/IBM/WebSphere/eXtremeScale/ObjectGrid/console/private/expanded/ibm/rainmaker.
acl-1.0.0.2/sql/derby/create-1.0.0.0.sql...
```

```

[info]Finished executing file
/usr/IBM/WebSphere/eXtremeScale/ObjectGrid/console/private/expanded/ibm/rainmaker.
acl-1.0.0.2/sql/derby/create-1.0.0.0.sql.
removing cache for user because of an event user-role-added
CWZC01005I: SMTP server has not been configured - email to will not be sent.
removing cache for user because of an event user-role-added
removing cache for user because of an event user-role-added
removing cache for user because of an event user-role-added
removing cache for user because of an event user-role-added
Application started and servicing requests at http://localhost:7080/
https://localhost:7443/
CWPZT0600I: Command start was successful

```

The solution is now published as a web doc at this website:

<http://www-01.ibm.com/support/docview.wss?uid=swg21502255&wv=1>

Special installation instructions for Solaris

You can run the web console on Solaris, although it is not officially supported. Example 7-27 shows the instructions to run the web console on Solaris.

Example 7-27 Instructions for running the web console on Solaris

```

#####
#
# Instructions to run IBM WebSphere eXtreme Scale (XS) V7.1.0 stand alone console
# AS-IS in an unsupported Solaris 64-bit environment.
#
#
# While these instructions will allow you to run the console on Solaris 64-bit
# the environment is still unsupported and support will be limited to issues
# that can be recreated on a supported OS platform only.
#
# The instuctions include 6 steps. Steps 4-6 include modifying IBM scripts.
# These modifications are already made in the scripts included with this read me.
#
# Files include with this readme:
# *startConsoleServer.sh
# *stopConsoleServer.sh
# *zero
#
# Note: If you choose to use the provided scripts instead of modifying your own
# please make sure the scripts have executable privileges.
#
#####

#####
#
# Step 1: Install full XS V7.1.0 stand alone console on a supported
# OS platform.
#
#####

#####
#
# Step 2: Install full XS V7.1.0 stand alone on Solaris 64-bit.

```

```

#
#####

#####
#
# Step 3: Copy the follow files from the supported XS to the Solaris 64-bit XS
# install.
#
#     1. Entire <install_root>/ObjectGrid/console directory
#     *2. <install_root>/ObjectGrid/bin/startConsoleServer.sh
#     *3. <install_root>/ObjectGrid/bin/stopConsoleServer.sh
#
# *Note: If using the provided scripts with this readme. Copy over the provided
#       start and stop console server scripts instead.
#
#####

#####
#
# Step 4: Modify new Solaris 64-bit file startConsoleServer.sh
#
#     1. Change from: /bin/sh zero start
#           to       : /bin/sh zero run
#
# Note: This change makes the console run outside the confines of a normal zero
#       process. Because of this we lose some controls which cause some benign
#       exceptions and messages to be shown in the command prompt after you
#       start the console. These can be ignored. Also some messages and
#       exceptions will continue to be outputted to the command prompt you start
#       the console as well as the typical log files. You can ignore what you see
#       in the command prompt or close it all together after starting.
#
#####

#####
#
# Step 5: Modify new Solaris 64-bit file stopConsoleServer.sh
#
#     1. Change from: /bin/sh $binDir/./console/zero stop
#           to       : cd $binDir/./console
#                   /bin/sh zero stop
#
#####

#####
#
# Step 6: Modify new Solaris 64-bit file <install_root>/ObjectGrid/console/zero
#
#     1. Change all instances of: if [ "$APP_HOME" -ef "/" ]; then
#           to                   : if [ "$APP_HOME" -ne "/" ]; then
#
#####


#####
#

```

```
# Done!
#
# Use the new startConsoleServer.sh and stopConsoleServer.sh as you would in a
# supported environment.
#
#####
```

Starting and using the console

We now briefly explain the steps to get the WebSphere eXtreme Scale web console up and running:

1. Start the web console by running the following command from the `WXS_install/ObjectGrid/bin` directory:
`startConsoleServer.sh`
 This command launches the web console server. After the command returns, the console server is running on the machine.
2. The web console server listens on port 7080, by default. Enter the following URL in a web browser to access the console:
`http://localhost:7080`
3. Log in with the user name and password `admin`. This password can be changed later from the console after the first login.
4. Now, configure the web console to use your catalog server. You need the host name and listener port for your catalog server:
 - a. In the console, click **Settings** → **eXtreme Scale Catalog Servers**.
 - b. Click the  icon to add a catalog server. Provide the host name and listener port for your catalog server, for example:
 - Host Name: `sa-w1201nx1.itso.ral.ibm.com`
 - Listener Port: `9809`

Stand-alone or managed catalog server: There is a slight difference in configuring a stand-alone catalog server or managed catalog server on the console:


► Stand-alone catalog server

For stand-alone servers, the default Java Management Extensions (JMX) port for the catalog server is 1099. You can change these settings by passing the parameters `-JMXServicePort` and `-listenerPort` during the start-up of the catalog server. See “Starting a catalog service domain on stand-alone JVMs” on page 139 for more information about starting the catalog server.

► Managed catalog server

A catalog server can run in a WebSphere Application Server process, and it is typically placed in a deployment manager or node agent process. For a catalog server that is hosted in WebSphere Application Server, the listener port is the bootstrap port of the catalog server process. (The deployment manager defaults to 9809 and the node agents default to 2809.)

5. Now, we need to configure a catalog service domain in the console:
 - a. Click **Settings** → **eXtreme Scale Domains**.

- b. Click the  icon to create a new catalog service domain. Provide an arbitrary name and the catalog server.
6. The console now is connected to your catalog server.

Tip: If the console does not connect to a catalog server, check the connectivity on the port and make sure that the catalog server is up and running. You can confirm that the catalog server is up and running by running a simple **xsadmin** command using the same host name and listener port:

```
WXS install\ObjectGrid\bin\xsadmin.sh -containers -ch
sa-w1201nx1.itso.ral.ibm.com -p 9809 -dmgr
```

7. You are now ready to use the console and view the reports on the monitoring tab. Figure 7-5 shows the current server overview from the web console.

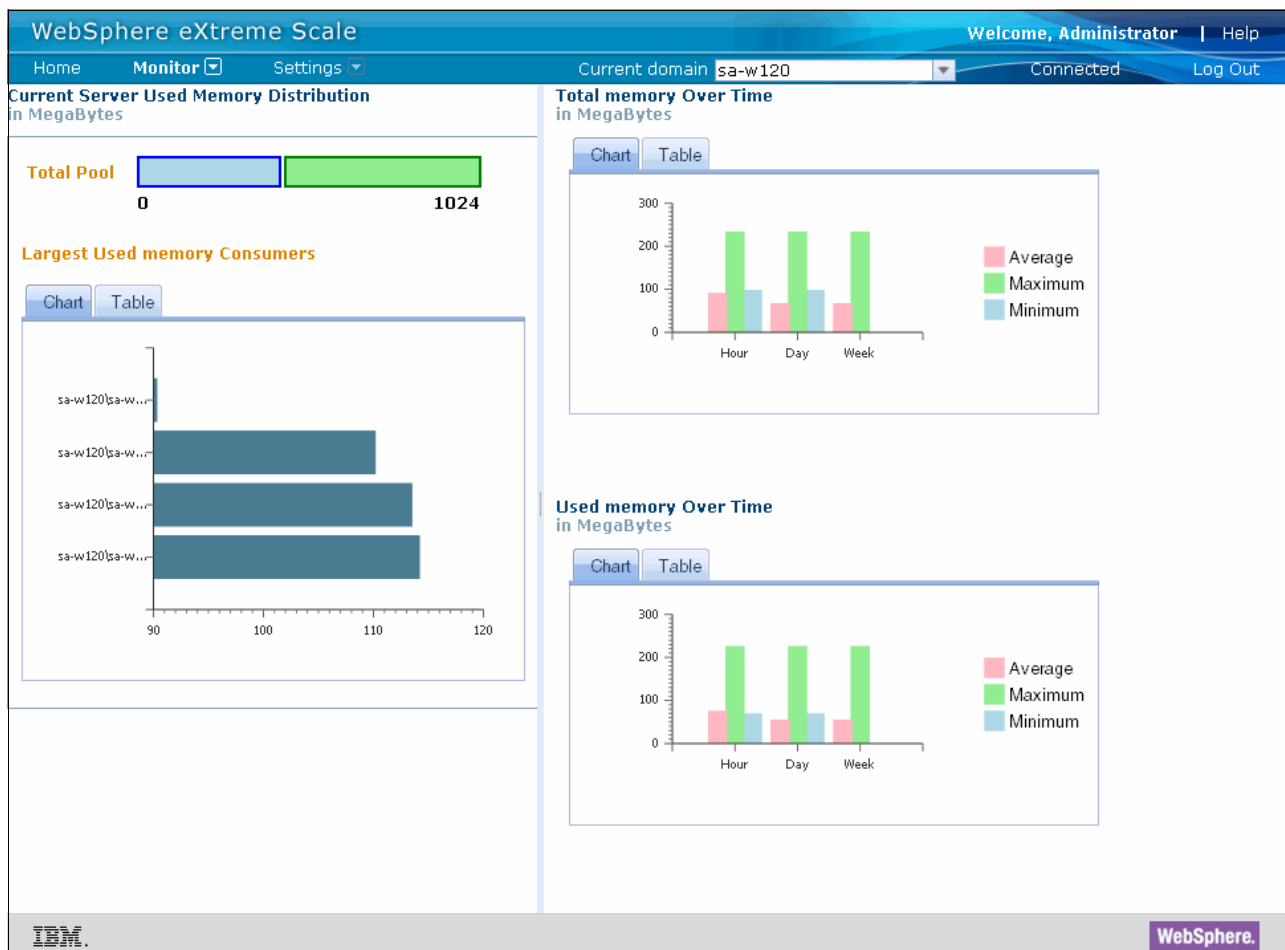


Figure 7-5 WebSphere eXtreme Scale web console: Server overview

For further information about the web console, see the information center:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.websphere.extremescale.admin.doc/txsmonitoroversw.html>

7.5.4 Monitoring with Tivoli Performance Viewer

The Tivoli Performance Viewer is a built-in monitoring tool that is available from the WebSphere administrative console. Administrators typically use it to monitor Performance Monitoring Infrastructure (PMI) metrics. Many runtime components in WebSphere Application Server can generate PMI metrics, for example, thread pools or Java Database Connectivity (JDBC) data source connection pools.

Production environments: The Tivoli Performance Viewer is well suited for monitoring in a production environment; however, the user interface tends to slow down when monitoring larger sets of data. Logging statistics over a longer period of time is also not possible. For production environments, consider the use of separate monitoring tools (Tivoli or third-party tools).

WebSphere eXtreme Scale PMI metrics

WebSphere eXtreme Scale can generate PMI metrics, which provide an insight into the server run time. The following types of information are available:

- ▶ Sizing: The number of entries in the map and the size of a map
- ▶ Cache statistics: Includes cache retrievals and hit rates
- ▶ Performance: Retrieval response times

Monitoring stand-alone environments: You can use PMI to monitor your environment only when you are using WebSphere eXtreme Scale with WebSphere Application Server. If you have a stand-alone deployment of WebSphere eXtreme Scale, you need to use the WebSphere eXtreme Scale web console (see 7.5.3, “WebSphere eXtreme Scale web console” on page 171).

Enabling ObjectGrid and ObjectMap PMI modules

You can use the Tivoli Performance Viewer to examine PMI statistics from only those modules that are enabled. The modules for the WebSphere eXtreme Scale PMI statistics (ObjectGrid and ObjectMap) are not enabled, by default.

Perform the following steps using the WebSphere administrative console to enable the PMI modules ObjectGrid and ObjectMap. Repeat the steps for each of the grid cluster members for which you want to view the PMI statistics:

1. In the left panel, click **Servers** → **WebSphere application servers**.
2. Click one of the WebSphere Application Server cluster members hosting the grid, for example, `sa-w1201nx2_container1`.
3. In the right panel under Performance, click **Performance Monitoring Infrastructure (PMI)**.
4. Click the **Runtime** tab.

Runtime tab: The benefit of making the changes under the Runtime tab is that they become effective immediately; however, these configuration changes are lost after a restart of the WebSphere Application Server process. You can avoid losing the configuration changes by making the changes under the Configuration tab, although this action requires a restart.

5. Confirm that **Enable Performance Monitoring Infrastructure (PMI)** is enabled and click **Custom**.

- Again, make sure to choose the **Runtime** tab.
- Click **ObjectGrid Maps** from the list of PMI modules. Select all the ObjectGrid Map counters that are listed in the panel on the right and click **Enable**. The counter status will change to Enabled, as shown in Figure 7-6.

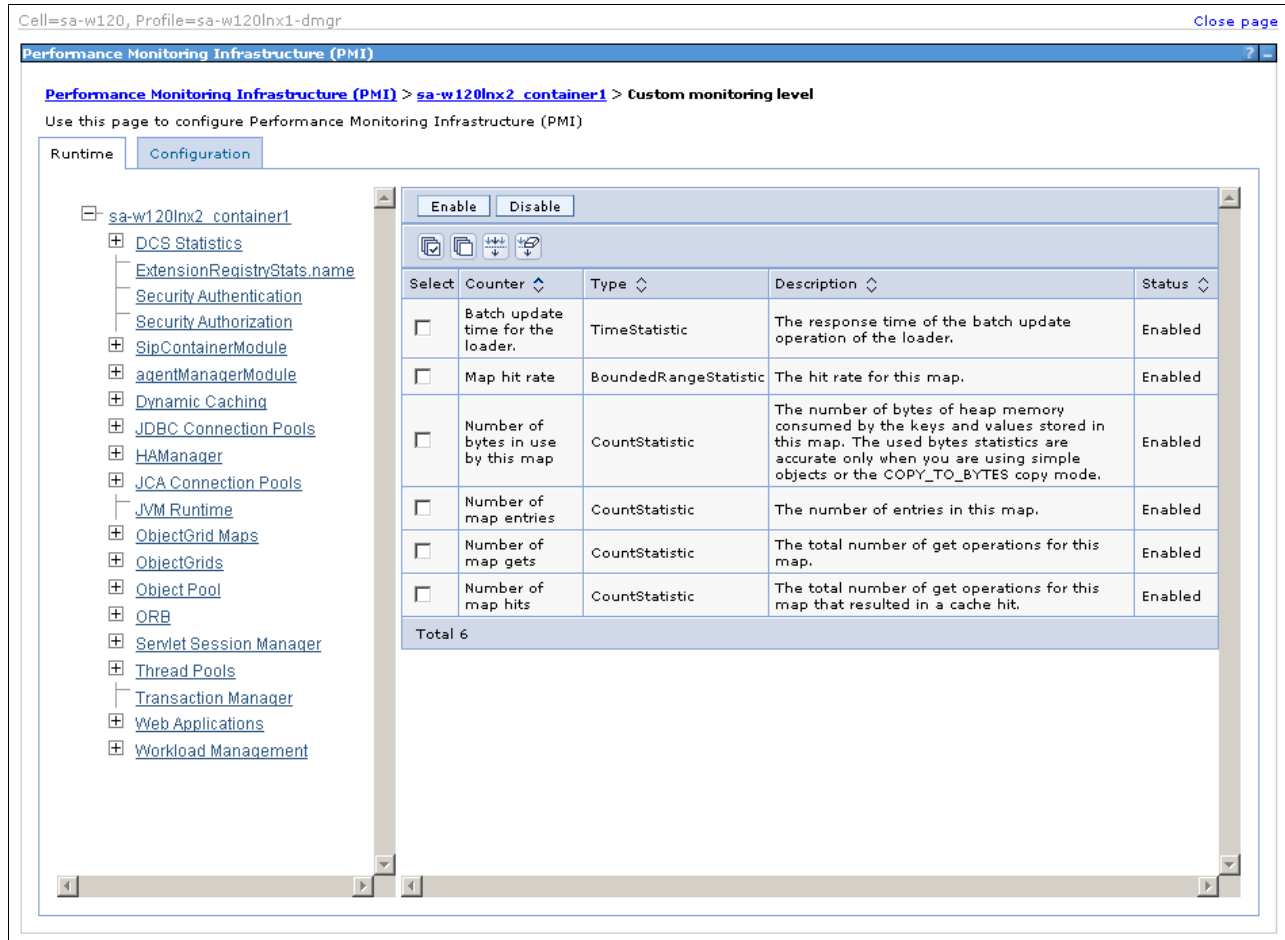


Figure 7-6 Enable counters for the ObjectGrid Maps PMI module

- Now, click **ObjectGrids** from the list of PMI modules. Select the ObjectGrid counter (there is only one) in the panel on the right and click **Enable**.

After the PMI modules for WebSphere eXtreme Scale have been enabled, you can monitor the PMI statistics in Tivoli Performance Viewer. Perform the following steps using the WebSphere administrative console to monitor the PMI statistics in Tivoli Performance Viewer:

- In the right panel, click **Monitoring and Tuning** → **Performance Viewer** → **Current Activity**.
- Select the grid cluster servers that you want to monitor and click **Start Monitoring**.
- Click a monitored server to open the Tivoli Performance Viewer for that server.
- Expand **Performance Modules** on the left side. Select the modules under ObjectGrids and ObjectGrid Maps that you want to view.
- When the view refreshes, the performance statistics display, as shown in Figure 7-7 on page 180.



Figure 7-7 Viewing ObjectGrid Maps PMI statistics using Tivoli Performance Viewer

7.5.5 Monitoring using a WebSphere eXtreme Scale ping client

In addition to the standard monitoring techniques that we have described, consider using a simple stand-alone WebSphere eXtreme Scale client for monitoring purposes. This client effectively connects to the WebSphere eXtreme Scale grid on a regular basis to perform a number of simple create, retrieve, update, and delete (CRUD) operations. If the WebSphere

eXtreme Scale grid is not working, this client fails to complete those operations. This stand-alone ping client can be deployed and operated from a cron job on a UNIX or AIX platform.

7.5.6 Additional monitoring tools

WebSphere eXtreme Scale can be integrated with several popular enterprise monitoring solutions. Plug-in agents are included for IBM Tivoli Monitoring and Hyperic HQ, which monitor WebSphere eXtreme Scale using publicly accessible management beans. CA Wily Introscope uses Java method instrumentation to capture statistics.

IBM Tivoli Enterprise Monitoring Agent for WebSphere eXtreme Scale

The IBM Tivoli Enterprise Monitoring Agent is a feature-rich monitoring solution that you can use to monitor databases, operating systems, and servers in distributed and host environments. WebSphere eXtreme Scale includes a customized agent that you can use to introspect WebSphere eXtreme Scale management beans. This solution works effectively for both stand-alone and WebSphere Application Server deployments.

CA Wily Introscope

CA Wily Introscope is a third-party management product that you can use to detect and diagnose performance problems in enterprise application environments. You can configure CA Wily Introscope to introspect selected portions of the WebSphere eXtreme Scale process to quickly view and validate WebSphere eXtreme Scale applications. CA Wily Introscope works effectively for both stand-alone and WebSphere Application Server deployments.

Hyperic HQ

Hyperic HQ is a third-party monitoring solution that is available at no cost as an open source solution or as an enterprise product. WebSphere eXtreme Scale includes a plug-in that allows Hyperic HQ agents to discover WebSphere eXtreme Scale container servers and to report and aggregate statistics using WebSphere eXtreme Scale management beans. You can use Hyperic HQ to monitor stand-alone WebSphere eXtreme Scale deployments.

For more information about using CA Wily Introscope and Hyperic HQ, see *Monitoring with vendor tools* at this website:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp?topic=/com.ibm.webSphere.extremescale.admin.doc/cxsmonitorvendor.html>

7.6 Applying product updates

Just like any other piece of enterprise software, WebSphere eXtreme Scale needs to be updated from time to time, for example, by upgrading from V7.0 to V7.1 or applying a fix pack or interim fix. In a production environment, it is vital to understand the impact of applying updates and maintaining software. Minimizing the interruption of the WebSphere eXtreme Scale grid is desirable. In certain cases, a loss of the grid for a brief period of time can have a big impact, particularly when dealing with grids that hold large data sets.

WebSphere eXtreme Scale is designed with these concepts in mind. It is possible to apply software maintenance to the product *without* interrupting the service. Although, a number of conditions need to be met in order to apply software maintenance to the product without interrupting the service.

7.6.1 Overview

When applying maintenance to WebSphere eXtreme Scale, it is important to realize that three logical components are involved. To avoid interrupting service, you *must* apply maintenance in the following order:

1. Catalog servers
2. Containers
3. Clients

The catalog servers must always be at the highest software level. Avoid situations where several of the container servers (or clients) are interacting with back-level catalog servers.

Topologies where catalog servers and containers are deployed on separate machines, are optimal for maintenance purposes. Maintenance can be applied to catalog servers without impacting the containers, because each has its own binary installation on its own machine.

Alternative: Although not recommended, it is possible to use a topology where catalog servers and containers are colocated. But, applying maintenance to the catalog server without impacting the container becomes more difficult. All running components that use the same set of binaries have to be stopped, because they each hold locks on the jars and libraries.

In this case, consider using two separate WebSphere eXtreme Scale installations. This design allows both the catalog server and the container to use their own sets of binaries. Applying maintenance to one no longer affects the other.

7.6.2 Procedures

See the product documentation for a detailed description of how to apply product updates to the WebSphere eXtreme Scale environment without interrupting service:

<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/topic/com.ibm.websphere.extramescale.admin.doc/txsupdateserv.html>

Upgrade procedures in an WebSphere eXtreme Scale environment include the following tasks:

- ▶ Operating system (OS) upgrade
- ▶ WebSphere eXtreme Scale product upgrade
- ▶ Applying a patch to an application, which is a grid client, in two forms:
 - The patch includes no change to the objects that are stored in the grid (or logic changes only, no data changed).
 - The patch includes changes to the objects that are stored in the grid.

Using application-specific properties files: The procedures in this section make use of application-specific properties files. Specifically, all applications have their grid names passed to them in a properties file that can be edited, if necessary. This method is perfectly acceptable.

Service scenarios and their recommended procedures

Next, we describe the various service scenarios. For each scenario, we describe the recommended procedure, referring to the typical pre-WebSphere eXtreme Scale procedure

when appropriate, and describing how it changes or is enhanced with WebSphere eXtreme Scale in your world.

Operating system upgrade

With a good topology, you only have to bring the JVMs down and back up for an operating system upgrade. This process is the same process that you use to recycle WebSphere Application Server or Tomcat application server JVMs when upgrading the OS on an application server box. You can use a rolling restart consistent with having enough catalogs and containers up for the grid to continue functioning.

WebSphere eXtreme Scale product upgrade

Because of the IBM commitment to backward compatibility, you can apply WebSphere eXtreme Scale upgrades (fix packs or new versions) without taking the grid down (although, of course, it is simpler if you can). This capability includes upgrading the WebSphere eXtreme Scale product to a new version or fix pack level or changing from 32-bit to 64-bit or back.

The process for upgrading (and for backing out an upgrade) consists of multiple steps, but it is straight-forward. The process assumes two hosts, N1 and N2, with the catalog servers on the same hosts as the container servers. The process is simpler if the catalog servers are on their own hosts. This process also assumes that N1 catalog servers use separate WebSphere eXtreme Scale binaries from the N1 container servers, and likewise for N2. You use separate binaries to avoid having .jar file locks that are held by still-running JVMs from interfering with the upgrades. If catalog servers are on their own hosts and are split across two hosts, you already have separate binaries.

Follow this process to apply an upgrade:

1. Stop N1 catalogs.
2. Stop N1 containers. The topology ensures that the grid can still function if no further significant failures occur.
3. Upgrade N1.
4. Start N1 catalogs.
5. Stop N2 catalogs.
6. Start N1 containers. They access the upgraded catalogs on N1.
7. Stop N2 containers. This topology ensures that the grid can still function if no further significant failures occur.
8. Upgrade N2.
9. Start all on N2.

The upgrade is now complete.

Follow this process to back out an upgrade:

1. Stop the catalogs and containers on N2. This action ensures that the grid can still function if no further significant failures occur.
2. Back out the upgrade on N2.
3. Start the N2 containers.
4. Stop the N1 containers. This action ensures that the grid can still function if no further significant failures occur.
5. Start the N2 catalogs. This step is safe, because only the prior version containers on N2 are running.

6. Stop the N1 catalogs.
7. Back out the upgrade on N1.
8. Start all the N1 containers.

Application patch with no changes to objects stored in the grid

This patch is the same process as with any application. Perform a rolling restart of your client JVMs or WebSphere Application Server client JVMs.

Application patch with changes to objects stored in the grid

This application patch with changes to objects that are stored in the grid is also known as “*object-schema change*”.

Another option: Although the process that is outlined in this section will work, you might decide that you prefer the simpler but more intrusive option of routing traffic away from one site at a time and upgrading that site. A consequence of this option is that a given site must be fully on Version *X* or Version *X+1*. A mixed mode on a single site cannot be done unless you use the EvenGrid/OddGrid technique that is described in this section.

This type of patch is not a common situation with most applications that are being developed. This situation is problematic, because there can be two versions of your application running at once; one version can write an object to the grid and the other version can read the object. The object read can be older than the reader-application’s version. The object read can also be newer than the reader’s version, which is even more difficult to properly handle. The following is the recommended process to handle this situation. We call this process “*EvenGrid/OddGrid*”.

In the EvenGrid/OddGrid process, each application (or application set) can be configured to use one of two grids, named, for example, EvenGrid and OddGrid (when we use “grid” here, we refer to a cluster of container JVMs). These two grids share the same catalog server cluster and can share the catalog server cluster with other application families and their even/odd pair of grids. Because each application family has its own pair of grids, they are actually called something similar to “App1_EvenGrid” and “App1_OddGrid”, but we will keep the names simple for this example.

EvenGrid and OddGrid are hosted on the same hardware and have the same container cluster structure. There is an EvenGrid container cluster whose JVMs host EvenGrid and the same for the OddGrid container cluster. The same WebSphere eXtreme Scale XML files, properties files, and so forth are used to start both the EvenGrid container cluster and the OddGrid container cluster. The only difference is the grid name. Normally, only one grid (one container cluster) is started and running. When you have a new application version to install, which includes an object-schema change, perform the following steps (assume that EvenGrid is currently running):

1. Start the OddGrid container cluster of JVMs. They come up empty and, at the moment, no application JVM is accessing them.
2. Change the property file that is used by the application to look up its GRIDNAME, changing that value to OddGrid. This change does not affect any currently running application JVMs, because the WebSphere eXtreme Scale preferred practice is to look up your grid on the application JVM start-up and save it for later reuse.
3. Perform a rolling restart of your application JVMs. As each application JVM restarts, it will reread the GRIDNAME value and reconnect to the grid, but it will be connected to OddGrid this time. JVMs not yet recycled will still be connected to EvenGrid. In this way, the restarted JVMs, which are running the new application version, will also use OddGrid

to store objects, while the unrecycled JVMs, which are running the old application version, will use EvenGrid to store objects. This way, there is no conflict between the old object format and the new object format.

4. When all application JVMs are restarted, no one is using EvenGrid any longer, and its container JVMs can be stopped.

There is a downside to this option if it is done during a busy period. During the transition process, you can use twice your planned memory. However, most clients can arrange that this upgrade does not occur during a busy period. The grid memory capacity is sized to handle a peak load and able to easily handle twice the normal memory use at 2 a.m.

During a slow period, you can use both grids if necessary. If an object existed in the old grid but was asked for from the new grid, a null will be returned (a “*cache miss*”). However, this situation is alright, because a cache miss can be returned during normal day-to-day processing. In which case, the applications must be able to handle it anyway, for example, by going to the database to recreate the object from a table row.

There are no other downsides to this option; all applications can share the same catalog server cluster and thus keep the operation's complexity to a minimum. No special application code change is required. The same application-specific WebSphere eXtreme Scale configuration is used for both EvenGrid and OddGrid, only the GRIDNAME value changes.

It is required that the applications follow the leading practice of getting the grid reference at start-up and saving it. This approach has performance benefits and you need to use this approach anyway.

The application upgrade is now complete. The EvenGrid is stopped. The OddGrid is currently running and will remain running until the next application with an upgrade that changes the object schema is required.



A

Sample code

This appendix contains sample code that is referenced in the book.

FastSerializabledKeyOrEntry_Externalizable.java

Example A-1 shows the code for the FastSerializabledKeyOrEntry_Externalizable.java file that is discussed in “Use Externalizable for efficient serialization” on page 129.

Example A-1 Code for the FastSerializabledKeyOrEntry_Externalizable.java file

```
/**
 *
 */
package com.ibm.issf.atjolin.serialization;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;

/**
 * @author Art Jolin, IBM Corp
 *
 */
public class FastSerializabledKeyOrEntry_Externalizable implements Externalizable
{
    private static final long serialVersionUID = 1L;

    // Instance variables (that is, fields). All object fields are optional
    (a.k.a., nullable) except shortNeverNullField and collectionField
    private int intField;
    private int[] intArrayField;
    private Short shortNeverNullField;
    private Long longObjectField;
    private String stringField; // Note special use of "intern()" method in setter
    and equals()
    private MyTestExternalizableObject myTestExternalizableObject;
    private MyTestObject customObjectField;
    private ArrayList<MyTestExternalizableObject> externalizableCollectionField;
    private ArrayList<MyTestExternalizableObject>
    nullableExternalizableCollectionField;
    private HashMap<MyTestExternalizableObject, MyTestExternalizableObject>
    externalizableMapField;
    private HashMap<MyTestExternalizableObject, MyTestExternalizableObject>
    nullableExternalizableMapField;

    private transient String transientStringField;
    private transient int fHashCode; // not part of this class's data, used in
    support of the hashCode method
    // as it is faster if we don't recalculate on every call

    /**
     * FastSerializabledKeyOrEntry_Externalizable - an example class showing how a
     fast serializable class
    */
}
```

```

    * can be written so it is usable in a distributed architecture (for example,
one using
    * WebSphere Extreme Scale, a.k.a. ObjectGrid). Note that some methods are
optional for entry
    * classes (the "value" part of a key/value pair) but are required for key
classes. There is never
    * any harm in doing them all even for entry classes and may save you trouble
later.
    */
    public FastSerializableKeyOrEntry_Externalizable() {

        /******* Serialization support and related *****/
        /* (non-Javadoc)
        * @see java.io.Externalizable#writeExternal(java.io.ObjectOutput)
        */
        @Override
        public void writeExternal(ObjectOutput out) throws IOException {
            // Write the serialVersionUID so we include what version this object was.
This allows one side of an app
            // to send an object of version X and have it deserialized into a version Y
object. This allows for rolling
            // update of your application when an object changes its fields.
            out.writeLong(serialVersionUID);

            out.writeInt(intField);
            SerializationUtils.writeIntArray(out, intArrayField);
            out.writeShort(shortNotNullField.shortValue());
            SerializationUtils.writeLong(out, longObjectField);
            SerializationUtils.writeUTF(out, stringField);
            SerializationUtils.writeNullableExternalizable(out,
myTestExternalizableObject);
            SerializationUtils.writeNullableObject(out, customObjectField);
            SerializationUtils.writeExternalizableCollection(out,
externalizableCollectionField);
            SerializationUtils.writeNullableExternalizableCollection(out,
nullableExternalizableCollectionField);
            SerializationUtils.writeExternalizableMap(out, externalizableMapField);
            SerializationUtils.writeNullableExternalizableMap(out,
nullableExternalizableMapField);
        }

        /* (non-Javadoc)
        * @see java.io.Externalizable#readExternal(java.io.ObjectInput)
        */
        @Override
        public void readExternal(ObjectInput in) throws IOException,
            ClassNotFoundException {
            // Read the serialVersionUID and use the value to decide how to interpret
this stream and turn it into an object
            //of our current version (the value of this.serialVersionUID).
            long incomingSerialVersionUID = in.readLong();
            // if (incomingSerialVersionUID == this.serialVersionUID) {...}

            intField = in.readInt();

```

```

        intArrayField = SerializationUtils.readIntArray(in);
        shortNeverNullField = in.readShort();
        longObjectField = SerializationUtils.readLong(in); //Java's automatic
        "wrapper" conversion from long to Long happens here
        stringField = SerializationUtils.readUTF(in).intern();

        //NOTE: we have to create an "empty" externalizable object so
        readNullableExternalizable(...) has something to fill in (it doesn't know about
        your object types & their constructors)
        // HOWEVER, we have to set our field to what readNullableExternalizable(...)
        returns, as it may return a null.
        myTestExternalizableObject = new MyTestExternalizableObject();
        myTestExternalizableObject =
        (MyTestExternalizableObject)SerializationUtils.readNullableExternalizable(in,
        myTestExternalizableObject);

        customObjectField = (MyTestObject)SerializationUtils.readNullableObject(in);
        try {
            externalizableCollectionField =
            (ArrayList)SerializationUtils.readExternalizableCollection(in, ArrayList.class,
            MyTestExternalizableObject.class);
            nullableExternalizableCollectionField =
            (ArrayList)SerializationUtils.readNullableExternalizableCollection(in,
            ArrayList.class, MyTestExternalizableObject.class);
            externalizableMapField = (HashMap)
            SerializationUtils.readExternalizableMap(in, HashMap.class,
            MyTestExternalizableObject.class, MyTestExternalizableObject.class);
            nullableExternalizableMapField = (HashMap)
            SerializationUtils.readNullableExternalizableMap(in, HashMap.class,
            MyTestExternalizableObject.class, MyTestExternalizableObject.class);
        } catch (IllegalAccessException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (InstantiationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    /*
     * hashCode - this method must be implemented for classes used as keys in
    ObjectGrid
     * @see java.lang.Object#hashCode()
     */
    public int hashCode() {
        // It would be much simpler if this object had some one field that was
        unique...but we don't.
        // If you are a RAD user, you can also generate the code for this method.

        // If this object was immutable we could cache the hashCode value. Since we
        are not
        //      (e.g., we have setter methods) you see certain lines commented
        out here.
        //

```

```

//    if (fHashCode == 0) {
//        fHashCode = hashCodeUtils.SEED;
//        //collect the contributions of various fields
//        fHashCode = hashCodeUtils.hash(fHashCode, intField);
//        fHashCode = hashCodeUtils.hash(fHashCode, intArrayField);
//        fHashCode = hashCodeUtils.hash(fHashCode,
shortNeverNullField.shortValue()); //you could just pass the Short itself, I prefer
this way for primitive wrappers
//        fHashCode = hashCodeUtils.hash(fHashCode, longObjectField); // this Long
could be null, so I decided to pass the object itself
//        fHashCode = hashCodeUtils.hash(fHashCode, stringField);
//        fHashCode = hashCodeUtils.hash(fHashCode, myTestExternalizableObject);
//        fHashCode = hashCodeUtils.hash(fHashCode, customObjectField);
//        fHashCode = hashCodeUtils.hash(fHashCode, externalizableCollectionField);
//        fHashCode = hashCodeUtils.hash(fHashCode,
nullableExternalizableCollectionField);
//        fHashCode = hashCodeUtils.hash(fHashCode, externalizableMapField);
//        fHashCode = hashCodeUtils.hash(fHashCode,
nullableExternalizableMapField);
//        fHashCode = hashCodeUtils.hash(fHashCode, transientStringField); //
transients are still data, in case you store objects locally they should be part
of the hashCode() value
//    }
//    return fHashCode;
//}

/*
 * equals - this method must be implemented for classes used as keys in
ObjectGrid
 * @see java.lang.Object#equals(java.lang.Object)
 */
public boolean equals(Object o) {
    FastSerializableKeyOrEntry_Externalizable castO = null;
    // First, if they are the same object just return immediately, for speed
    if (o == this) return true;

    // Next, if "o" is null then it cannot be equal to "this" so return false
    if (o == null) return false;

    // Another quick test: the hashCode() + equals() contract required by Java
    (see Object's javadoc) says
    // that if two objects are equivalent (o1.equals(o2) is true) then they must
    return the same value for hashCode().
    // Therefore, if "this" and "o" return DIFFERENT values for hashCode() then
    they MUST NOT be equal.
    // This is a quick test because we save the computed value of our hashcode in
    an instance variable, so hashCode() is
    // very very fast most of the time.
    //    if (this.hashCode() != o.hashCode()) return false;

    // At this point, "o" MAY be equivalent to "this", we have further checks to
    do.

    // Next, make sure "o" is the correct type, then cast it for later use
    if (o instanceof FastSerializableKeyOrEntry_Externalizable) {

```

```

        cast0 = (FastSerializableKeyOrEntry_Externalizable)o;
    } else {
        return false;
    }

    // Compare each of our fields, recursively. For objects, do quick == check
    first (which also catches if both are null and thus equal).
    // After the quick == we insure the field in cast0 isn't null, thus avoiding
    an NPE in the following equals(...) call. Lastly we
    // check equivalence with equals().
    // Arrays are handled with the Java helper method java.util.Arrays, which
    checks for same elements and in the same order, or both null.
    if (
        (cast0.intField == this.intField) &&
        Arrays.equals(cast0.intArrayField, this.intArrayField) &&
        ((cast0.shortNonNullField == this.shortNonNullField) ||
        ((cast0.shortNonNullField!=null) &&
        cast0.shortNonNullField.equals(this.shortNonNullField))) &&
        ((cast0.longObjectField == this.longObjectField) ||
        ((cast0.longObjectField!=null) &&
        cast0.longObjectField.equals(this.longObjectField))) &&
        (cast0.stringField.intern() == this.stringField) && //we can be quick like
        this because we intern'd in our setter also
        ((cast0.myTestExternalizableObject == this.myTestExternalizableObject) ||
        ((cast0.myTestExternalizableObject!=null) &&
        cast0.myTestExternalizableObject.equals(this.myTestExternalizableObject))) &&
        ((cast0.customObjectField == this.customObjectField) ||
        ((cast0.customObjectField!=null) &&
        cast0.customObjectField.equals(this.customObjectField))) &&
        ((cast0.externalizableCollectionField ==
        this.externalizableCollectionField) ||
        ((cast0.externalizableCollectionField!=null) &&
        cast0.externalizableCollectionField.equals(this.externalizableCollectionField)))
        &&
        ((cast0.nullableExternalizableCollectionField ==
        this.nullableExternalizableCollectionField) ||
        ((cast0.nullableExternalizableCollectionField!=null) &&
        cast0.nullableExternalizableCollectionField.equals(this.nullableExternalizableColl
        ectionField))) &&
        ((cast0.externalizableMapField == this.externalizableMapField) ||
        ((cast0.externalizableMapField!=null) &&
        cast0.externalizableMapField.equals(this.externalizableMapField))) &&
        ((cast0.nullableExternalizableMapField ==
        this.nullableExternalizableMapField) ||
        ((cast0.nullableExternalizableMapField!=null) &&
        cast0.nullableExternalizableMapField.equals(this.nullableExternalizableMapField)))
    ) {
        return true;
    } else {
        return false;
    }
}

public FastSerializableKeyOrEntry_Externalizable clone() {

```

```

        FastSerializableKeyOrEntry_Externalizable newClone = new
FastSerializableKeyOrEntry_Externalizable();
        newClone.intField = this.intField;
        newClone.intArrayField = this.intArrayField;
        newClone.shortNonNullField = this.shortNonNullField;
        newClone.longObjectField = this.longObjectField;
        newClone.stringField = this.stringField;
        newClone.myTestExternalizableObject = this.myTestExternalizableObject;//
this example isn't doing a "deep" copy (where we would create new contained
objects with the same data)
        newClone.customObjectField = this.customObjectField;
        newClone.externalizableCollectionField = this.externalizableCollectionField;
        newClone.nullableExternalizableCollectionField =
this.nullableExternalizableCollectionField;
        newClone.externalizableMapField = this.externalizableMapField;
        newClone.nullableExternalizableMapField =
this.nullableExternalizableMapField;

        newClone.transientStringField = this.transientStringField;
        newClone.fHashCode = 0;

        return newClone;
    }

    /******* Accessors *****/

    public int getIntField() {
        return intField;
    }

    public void setIntField(int intField) {
        this.intField = intField;
        this.fHashCode = 0;// Each setter (potentially) changes a value used in the
hashCode, so force a recalculation the next time hashCode() is called
        // (of course, for key objects you should NEVER call a
setter anyway).
    }

    public int[] getIntArrayField() {
        return intArrayField;
    }

    public Short getShortNonNullField() {
        return shortNonNullField;
    }

    public void setShortNonNullField(Short shortNonNullField) {
        this.shortNonNullField = shortNonNullField;
        this.fHashCode = 0;
    }

    public void setIntArrayField(int[] intArrayField) {
        this.intArrayField = intArrayField;
        this.fHashCode = 0;
    }

```

```

    }

    public Long getLongObjectField() {
        return longObjectField;
    }

    public void setLongObjectField(Long longObjectField) {
        this.longObjectField = longObjectField;
        this.fHashCode = 0;
    }

    public String getStringField() {
        return stringField;
    }

    public void setStringField(String stringField) {
        // calling "intern()" returns the one and only central copy of this String,
        which makes "equals()" faster
        this.stringField = stringField.intern();
        this.fHashCode = 0;
    }

    public MyTestExternalizableObject getMyTestExternalizableObject() {
        return myTestExternalizableObject;
    }

    public void setMyTestExternalizableObject(
        MyTestExternalizableObject myTestExternalizableObject) {
        this.myTestExternalizableObject = myTestExternalizableObject;
    }

    public MyTestObject getCustomObjectField() {
        return customObjectField;
    }

    public void setCustomObjectField(MyTestObject customObjectField) {
        this.customObjectField = customObjectField;
        this.fHashCode = 0;
    }

    public ArrayList<MyTestExternalizableObject> getExternalizableCollectionField()
    {
        return externalizableCollectionField;
    }

    public void setExternalizableCollectionField(
        ArrayList<MyTestExternalizableObject> externalizableCollectionField) {
        this.externalizableCollectionField = externalizableCollectionField;
    }

    public ArrayList<MyTestExternalizableObject>
    getNullableExternalizableCollectionField() {
        return nullableExternalizableCollectionField;
    }

```

```

        public void setNullableExternalizableCollectionField(
            ArrayList<MyTestExternalizableObject>
nullableExternalizableCollectionField) {
            this.nullableExternalizableCollectionField =
nullableExternalizableCollectionField;
        }

        public HashMap<MyTestExternalizableObject, MyTestExternalizableObject>
getExternalizableMapField() {
            return externalizableMapField;
        }

        public void setExternalizableMapField(
            HashMap<MyTestExternalizableObject, MyTestExternalizableObject>
externalizableMapField) {
            this.externalizableMapField = externalizableMapField;
        }

        public HashMap<MyTestExternalizableObject, MyTestExternalizableObject>
getNullableExternalizableMapField() {
            return nullableExternalizableMapField;
        }

        public void setNullableExternalizableMapField(
            HashMap<MyTestExternalizableObject, MyTestExternalizableObject>
nullableExternalizableMapField) {
            this.nullableExternalizableMapField = nullableExternalizableMapField;
        }

        public String getTransientStringField() {
            return transientStringField;
        }

        public void setTransientStringField(String transientStringField) {
            this.transientStringField = transientStringField;
            this.fHashCode = 0;
        }
    }
}

```



Additional material

This book refers to additional material that can be downloaded from the Internet as described in the following sections.

Locating the Web material

The web material associated with this book is available in softcopy on the Internet from the IBM Redbooks web server. Point your web browser at:

<ftp://www.redbooks.ibm.com/redbooks/SG247964>

Alternatively, you can go to the IBM Redbooks website at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG247964.

Using the Web material

The additional web material that accompanies this book includes the following files:

<i>File name</i>	<i>Description</i>
SerializationExamplesAndUtilities.jar	Contains serialization code examples
WAS+XS_ExternalXML_Package.zip	Contains sample eXtreme Scale code that allows you to start a WebSphere Application Server application server as an eXtreme Scale container using your XML files but with those files external to any compiled .ear or .war.

Downloading and extracting the Web material

Create a subdirectory (folder) on your workstation, and extract the contents of the Web material .zip file into this folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- ▶ *IBM WebSphere eXtreme Scale V7: Solutions Architecture*, REDP-4602
- ▶ *User's Guide to WebSphere eXtreme Scale*, SG24-7683
- ▶ *Scalable, Integrated Solutions for Elastic Caching Using IBM WebSphere eXtreme Scale*, SG24-7926
- ▶ *WebSphere Business Integration V6.0.2 Performance Tuning*, REDP-4304
- ▶ *WebSphere Application Server V7 Administration and Configuration Guide*, SG24-7615

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Online resources

These websites are also relevant as further information sources:

- ▶ WebSphere eXtreme Scale V7.1 Information Center
<http://publib.boulder.ibm.com/infocenter/wxsinfo/v7r1/index.jsp>
- ▶ WebSphere Application Server V7 Information Center
http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.nd.multiplatform.doc/info/welcome_nd.html
- ▶ Web Doc: SRVE0068E exception thrown in CORBA.OBJ_ADAPTER
<http://www-01.ibm.com/support/docview.wss?uid=swg21426334>
- ▶ Sizing the Java heap
http://publib.boulder.ibm.com/infocenter/javasdk/tools/index.jsp?topic=/com.ibm.java.doc.igaa/_1vg00014884d287-11c3fb28dae-7ff6_1001.html
- ▶ APAR: PM37461: Fix Thread Safety Issue when Running Same Query Concurrently
<http://www-01.ibm.com/support/docview.wss?uid=swg1PM37461>
- ▶ Web Doc: CWPZC8029E startConsoleServer.bat script fails on 64-bit Windows
<http://www-01.ibm.com/support/docview.wss?rs=3023&context=SSPPLQ&q1=v71xsrnotes&uid=swg21438057>

- ▶ *WebSphere eXtreme Scale Administration Guide*
ftp://ftp.software.ibm.com/software/webserver/appserv/library/v71/xSadminguide_PDF.pdf
- ▶ “How to do online runtime upgrades for IBM WebSphere eXtreme Scale” by Billy Newport
http://www.springone2gx.com/blog/billy_newport/2010/12/how_to_do_online_runtime_upgrades_for_ibm_websphere_extreme_scale

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services



WebSphere eXtreme Scale Best Practices for Operation and Management

Tips for capacity planning

Leading practices for operations

Grid configuration

This IBM Redbooks publication contains a summary of the leading practices for implementing and managing a WebSphere eXtreme Scale installation. The information in this book is a result of years of experience that IBM has had in with production WebSphere eXtreme Scale implementations. The input was received from specialists, architects, and other practitioners who have participated in engagements around the world.

The book provides a brief introduction to WebSphere eXtreme Scale and an overview of the architecture. It then provides advice about topology design, capacity planning and tuning, grid configuration, ObjectGrid and backing map plug-ins, application performance tips, and operations and monitoring.

This book is written for a WebSphere eXtreme Scale-knowledgeable audience.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks